Introduction

This book is about two software packages called Tcl and Tk.[1] Tcl is a *dynamic language* (also known as a *scripting language*) for controlling and extending applications; its name stands for “tool command language.” Tcl provides general programming facilities sufficient for most applications. Furthermore, Tcl is both *embeddable* and *extensible*. Its interpreter is a library of C functions that can easily be incorporated into applications, and each application can extend the core Tcl features with additional commands either unique to the application or provided by add-on libraries (referred to as *extensions* in the Tcl community).

1. The official pronunciation for Tcl is “tickle,” although “tee-see-ell” is also used frequently. Tk is pronounced “tee-kay.”

One of the most useful extensions to Tcl is Tk, which is a toolkit for developing graphical user interface (GUI) applications. Tk extends the core Tcl facilities with commands for building user interfaces, so that you can construct GUIs by writing Tcl scripts instead of C code. Like Tcl, Tk is implemented as a library of C functions so it can be used in many different applications.

Note

This book corresponds to Tcl/Tk version 8.5. The release notes for each version of Tcl/Tk describe the changes and new features in each release. The Tcler’s Wiki web site (`http://wiki.tcl.tk`) also compiles change lists for each release; you can find them by searching for pages that contain *Changes in* in the title.

I.1 Benefits of Tcl/Tk

Together, Tcl and Tk provide several benefits to application developers and users. The first benefit is rapid development. Many interesting applications

can be written entirely as Tcl scripts. This allows you to program at a much higher level than you would in C/C++ or Java, and Tk hides many of the details that C or Java programmers must address. Compared to low-level toolkits, there is much less to learn in order to use Tcl and Tk, and much less code to write. New Tcl/Tk users often can create interesting user interfaces after just a few hours of learning, and many people have reported tenfold reductions in code size and development time when they switched from other toolkits to Tcl and Tk.
Another reason for rapid development with Tcl and Tk is that Tcl is an interpreted language. When you use a Tcl application, you can generate and execute new scripts on the fly without recompiling or restarting the application. This allows you to test out new ideas and fix bugs rapidly. Since Tcl is interpreted, it executes more slowly than compiled C code; but internal optimizations, such as bytecode compilation coupled with ever-increasing processor power, have erased most of the perceived performance advantages of compiled languages. For example, you can execute scripts with hundreds of Tcl commands on each movement of the mouse with no perceptible delay. In the rare cases where performance becomes an issue, you can reimplement the performance-critical parts of your Tcl scripts in C.
A second benefit is that Tcl is a cross-platform language, as are most of its extensions, including Tk. This means that an application developed on one platform, such as Linux, in most cases can be run without change on another platform, such as Macintosh or Windows.
Tcl was also the first dynamic language to have native Unicode support. As a result, Tcl applications can handle text in virtually any of the world’s written languages. Tcl requires no extensions to process text in any of the Unicode-supported scripts, and standard extensions such as `msgcat` provide simple localization support.
Another significant benefit is that Tcl and most of its extensions are freely available as open source. Tcl and Tk follow the so-called BSD license, which allows anyone to download, inspect, modify, and redistribute Tcl/Tk without charge.
Tcl is an excellent “glue language.” A Tcl application can include many different extensions, each of which provides an interesting set of Tcl commands. Tk is one example of a library package; many other packages have been developed by the Tcl/Tk community, and you can also write your own packages. Tcl scripts for such applications can include commands from any of the packages.
Additionally, Tcl makes it easy for applications to have powerful scripting languages. For example, to add scripting capability to an existing

application, all you need do is implement a few new Tcl commands that provide the basic features of the application. Then you can link your new commands with the Tcl library to produce a full-function scripting language that includes both the commands provided by Tcl (called the *Tcl core*) and those that you wrote.
Tcl also provides user convenience. Once you learn Tcl and Tk, you will be able to write scripts for any Tcl and Tk application merely by learning the few application-specific commands for the new application. This should make it possible for more users to personalize and enhance their applications.

I.2 Organization of the Book

Chapter 1 uses several simple scripts to provide a quick overview of the most important features of Tcl and Tk. It is intended to give you the flavor of the systems and convince you that they are useful, without explaining anything in detail. The remainder of the book goes through everything again in a more comprehensive fashion. It is divided into three parts:

- **Part I** introduces the Tcl scripting language. After reading this section, you will be able to write scripts for Tcl applications. You will need to know at least some of the information in this part in order to write Tk applications.
- **Part II** describes the additional Tcl commands provided by Tk, which allow you to create user-interface widgets such as menus and scrollbars and arrange them in GUI applications. After reading this section, you will be able to create new GUI applications and write scripts to enhance existing Tk applications.
- **Part III** discusses the C functions in the Tcl library and how to use them to create new Tcl commands. After reading this section, you will be able to write new Tcl packages and applications in C. However, you will be able to do a great deal (perhaps everything you need) without this information.

Each of these parts contains about a dozen short chapters. Each chapter is intended to be a self-contained description of a piece of the system, and you need not necessarily read the chapters in order.
Not every feature of Tcl and Tk is covered here, and the explanations are organized to provide a smooth introduction rather than a complete reference source. A separate set of reference manual entries, referred to as the

reference documentation, is available with the Tcl and Tk distributions. These are much more terse but they cover absolutely every feature of both systems. Appendix A describes how to retrieve the Tcl and Tk distributions, including reference documentation, via the Internet. Appendix B provides a survey of some popular Tcl extensions. Appendix C lists additional online and printed resources for Tcl and Tk. The full Tcl Source Distribution License is included in Appendix D.

This book assumes that you already know how to use your operating system, including interacting with applications from the command line. Part III of this book assumes that you are familiar with the C programming language as defined by the ANSI C standard; a basic knowledge of C is helpful for Parts I and II but not required. You need not know anything about either Tcl or Tk before reading this book; both are introduced from scratch.

I.3 Notation

This book uses a `monospace` font for anything that might be typed to a computer, such as Tcl scripts, C code, and names of variables, procedures, and commands. The examples of Tcl scripts use notation like the following:

```
  set a 44
⇒44
```

Tcl commands, such as `set a 44` in the example, appear in `monospace`; their results, such as *`44`* in the example, appear in italicized *`monospace`*. The ⇒ symbol before the result indicates that this is a normal return value. If an error occurs in a Tcl command, the error message appears in italicized *`monospace`*, preceded by a Ø symbol to indicate that this is an error rather than a normal return:

```
  set a 44 55
Ø wrong # args: should be "set varName ?newValue?"
```

When describing the syntax of Tcl commands, italicized Courier is used for formal argument names. An argument or group of arguments enclosed in question marks indicates that the arguments are optional. For example, the syntax of the `set` command is as follows:

set *varName* ?*newValue*?

This means that the word `set` must be entered verbatim to invoke the command, and *`varName`* and *`newValue`* are the names of `set`'s arguments; when invoking the command, you type a variable name instead of *`varName`* and a new value for the variable instead of *`newValue`*. The *`newValue`* argument is optional.

1. An Overview of Tcl and Tk

This chapter introduces Tcl and Tk with a series of scripts illustrating their main features. Although you should be able to start writing simple scripts after reading this chapter, the explanations here are not complete. The purpose of this chapter is to show you the overall structure of Tcl and Tk and the kinds of things they can do, so that when individual features are discussed in detail you'll be able to see why they are useful. All of the information in this chapter is revisited in more detail in later chapters, and several important aspects, such as the Tcl C interfaces, are not discussed at all in this chapter.

1.1 Getting Started

To invoke Tcl scripts, you must run a Tcl application. If Tcl is installed on your system, there should exist a simple Tcl shell application called `tclsh`, which you can use to try out some of the examples in this chapter. If Tcl has not been installed on your system, refer to Appendix A for information on how to obtain and install it.

Note

It's common practice to install `tclsh` with its version number as part of the name (for example, `tclsh8.5` on Unix or `tclsh85` on Windows). This has the advantage of allowing multiple versions of Tcl to exist on the same system at once, but also the disadvantage of making it harder to write scripts that start uniformly across different versions of Tcl. Therefore, most installers also commonly link or alias `tclsh` to the most recent version installed on the system. The same is true for the `wish` interpreter, described later. Therefore, unless you want to use a specific version of a Tcl application installed on your system, you should simply use `tclsh` or `wish`.

You can start the `tclsh` application by opening a terminal window on a Macintosh or Unix system, or a command prompt window on a Windows system, and then entering the command

```
tclsh
```

This causes `tclsh` to start in interactive mode, reading Tcl commands from the keyboard and passing them to the Tcl interpreter for evaluation. For starters, enter the following command at the `tclsh` prompt:

```
expr 2 + 2
```

`tclsh` prints the result (`4`) and then prompts you for another command.
This example illustrates several features of Tcl. Each command consists of one or more *words* separated by spaces or tabs (referred to as *whitespace* characters). In the example there are four words: `expr`, `2`, `+`, and `2`. The first word of each command is the name of the command to execute. The other words are *arguments* that are passed to the command for processing. `expr` is one of the core commands provided by the Tcl library, so it exists in every Tcl application. It concatenates its arguments into a single string and evaluates the string as an arithmetic expression.
Each Tcl command returns a result. If a command has no meaningful result, it returns an empty string. For the `expr` command the result is the value of the expression.
All values in Tcl have a string representation and may also have a more efficient internal representation. In this example, `expr`'s result is a numerical value that would have a binary integer or floating-point internal representation. The internal representation allows faster and more efficient processing of information. If the value simply is assigned to a variable or if it is used by another command expecting a numerical value, this is done very efficiently as no string conversion is required. Tcl automatically generates a string representation of a value on an as-needed basis—for example, when the value is displayed on the console.

Note

At the script development level, you can treat all values as strings; Tcl converts between the string and the internal representation automatically as needed. As you grow more familiar with Tcl, understanding what can cause conversions can help you avoid them, resulting in more efficient and faster code. In general, always being consistent in your treatment of a value (e.g., always using list commands to process lists, always using dictionary commands to process dictionaries, etc.) and avoiding unnecessary printing and other string manipulation of numeric, list, and dictionary values can go a long way in speeding up your code. For more information on the internal representation of values at the C language level, see Chapter 32.

From now on, we will use notation such as the following to describe examples:

```
  expr 2 + 2
⇒ 4
```

The first line is the command you enter and the second line is the result returned by the command. The ⇒ symbol indicates that the line contains a return value; the ⇒ is not actually printed out by `tclsh`. We will omit return values in cases where they aren't important, such as sequences of commands where only the last command's result matters.
Commands are normally terminated by newlines (typically the Enter or Return key on your keyboard), so each line that you enter in `tclsh` normally becomes a separate command. Semicolons also act as command separators, in case you wish to enter multiple commands on a single line. It is also possible for a single command to span multiple lines; you'll see how to do this later.
The `expr` command supports an expression syntax similar to that of expressions in ANSI C, including the same precedence rules and most of the C operators. Here are a few examples that you could enter in `tclsh`:

```
  expr 2 * 10 - 1
⇒ 19
  expr 14.1*6
⇒ 84.6
  expr sin(.2)
⇒ 0.19866933079506122
  expr rand()
⇒ 0.62130973004797
  expr rand()
⇒ 0.35263291623100307
  expr (3 > 4) || (6 <= 7)
⇒ 1
```

The first example shows the multiplication operator and how it has a higher precedence than subtraction. The second shows that expressions can contain real values as well as integer values. The next examples show some of the

built-in functions supported by `expr`, including the `rand()` function for generating random numbers between 0 and 1. The last example shows the use of the relational operators `>` and `<=` and the logical OR operator `||`. As in C, Boolean results are represented numerically with `1` for true and `0` for false.
To leave `tclsh`, invoke the `exit` command:

```
exit
```

This command terminates the application and returns you to your shell.

1.2 “Hello, World!” with Tk

Tcl provides a full set of programming features such as variables, loops, and procedures. It can be used by itself or with extensions that implement their own Tcl commands in addition to those in the Tcl core.
One of the more interesting extensions to Tcl is the set of windowing commands provided by the Tk toolkit. Tk’s commands allow you to create graphical user interfaces. Many of the examples in this book use an application called `wish` (“windowing shell”), which is similar to `tclsh` except that it also includes the commands defined by Tk. If Tcl and Tk have been installed on your system, you can invoke `wish` from your terminal or command prompt window just as you did for `tclsh`; it displays a small empty window on your screen and then reads commands from the console. Alternatively, if you have Tcl/Tk version 8.4 or later installed, you can invoke the `tclsh` application, and then use the command `package require Tk` to dynamically load the Tk extension.

Note

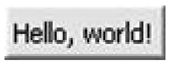
On Windows, invoking an interactive `wish` session displays both the empty window and a separate console window. The console window is a replacement for a real console to allow input and output on the standard I/O channels. The console window normally is hidden when a script file is executing, as described later, although you can display it by executing the `console show` command. Consult the `console` reference documentation for more information.

Here is a simple Tk script that you could run with `wish`:

```
button .b -text "Hello, world!" -command exit
grid .b
```

If you enter these two Tcl commands in `wish`, the window’s appearance changes to that shown in Figure 1.1. If you move the pointer over the “Hello, world!” text and click the main mouse button (the leftmost button in most configurations), the window disappears and `wish` exits.

Figure 1.1 The “Hello, world!” application



Several things about this example need explanation. First let us deal with the syntactic issues. The example contains two commands, `button` and `grid`, both of which are implemented by Tk. Although these commands do not look like the `expr` command in the previous section, they have the same basic structure as all Tcl commands: one or more words separated by whitespace characters. The `button` command contains six words, and the `grid` command contains two words.

The fourth word of the `button` command is enclosed in double quotes. This allows the word to include whitespace characters; without the quotes, `Hello,` and `world!` would be separate words. The double quotes are delimiters, not part of the word itself; they are removed by the Tcl interpreter before the command is executed.

For the `expr` command the word structure doesn’t matter much since `expr` concatenates all its arguments. However, for the `button` and `grid` commands, and for most Tcl commands, the word structure is important. The `button` command expects its first argument to be the name of a new window to create. Additional arguments to this command must come in pairs, where the first argument of each pair is the name of a *configuration option* and the second argument is a value for that option. Thus if the double quotes were omitted, the value of the `-text` option would be `Hello,` and `world!` would be treated as the name of a separate configuration option. Since there is no option defined with the name `world!` the command would return an error.

Now let us move on to the behavior of the commands. The basic building block for a graphical user interface in Tk is a *widget*. A widget is a window with a particular appearance and behavior (the terms *widget* and *window* are used synonymously in Tk). Widgets are divided into classes such as buttons, menus, and scrollbars. All the widgets in the same class have the same general appearance and behavior. For example, all button widgets display a text string, bitmap, or image and execute a Tcl script when the user clicks the button.

Widgets are organized hierarchically in Tk, with names that reflect their positions in the hierarchy. The *main widget*, which appeared on the screen when you started `wish`, has the name `.` and `.b` refers to a child `b` of the main widget. Widget names in Tk are like file name paths except that they use `.` as a separator character instead of `/` or `\`. Thus, `.a.b.c` refers to a widget that is a child of widget `.a.b`, which in turn is a child of `.a`, which is a child of the main widget.

Tk provides one command for each class of widgets, called a *class command*, which you invoke to create widgets of that class. For example, the `button` command creates button widgets. This is similar to standard object-oriented programming principles, though Tk doesn't support direct subclassing of the widget classes. All of the class commands have the same form: the first argument is the name of a new widget to create, and additional arguments specify configuration options. Different widget classes support different sets of options. Widgets typically have many options, with default values for the options that you don't specify. When a class command like `button` is invoked, it creates a new widget with the given name and configures it as specified by the options.

The `button` command in the example specifies two options: `-text`, which is a string to display in the button, and `-command`, which is a Tcl script to execute when the user invokes the button. In this example the `-command` option is `exit`. Here are a few other button options that you can experiment with:

- `-background`—the background color for the button, such as `blue`
- `-foreground`—the color of the text in the button, such as `black`
- `-font`—the font to use for the button, such as `"times 12"` for a 12-point Times Roman font

Creating a widget does not automatically cause it to be displayed. The `grid` command causes the button widget to appear on the screen. Independent entities called *geometry managers* are responsible for computing the sizes and locations of widgets and making them appear on the screen. The separation of widget creation and geometry management provides significant flexibility in arranging widgets on the screen to design your application. The `grid` command in the example asks a geometry manager called the *gridder* to manage `.b`. The gridder arranges widgets in a grid of columns and rows. In this case, the command placed `.b` in the first column of the first row of the grid and sized the grid to just large enough to accommodate the widget; furthermore, if the parent has more space than needed by the grid, as in the example, the parent is shrunk so that it is just large enough to hold the child. Thus, when you entered the `grid` command, the main window (.) shrank from its original size to the size that appears in Figure 1.1.

1.3 Script Files

In the examples so far, you have entered Tcl commands interactively to `tclsh` or `wish`. You can also place commands into script files and invoke the script files just like shell scripts. To do this for the "Hello, world!" example, place the following text in a file named `hello.tcl`:

```
#!/usr/local/bin/wish
button .b -text "Hello, world!" -command exit
pack .b
```

You can execute this script by invoking the `wish` interpreter and passing the script file name as a command-line argument:

```
wish hello.tcl
```

This causes `wish` to display the same window as shown in Figure 1.1 and wait for you to interact with it. In this case you will not be able to type commands interactively to `wish`; all you can do is click on the button.

1.3.1 Executable Scripts on Unix and Mac OS X

The script just shown is the same as the one you typed earlier except for the first line. As far as `wish` is concerned, this line is a comment, but on Unix systems if you make the file executable (for example, by executing `chmod +x hello.tcl` in your shell), you can then invoke the file directly by typing `hello.tcl` to your shell. (This requires the directory containing your `hello.tcl` script to be listed in your `PATH` environment variable.) When you do this, the system invokes `wish`, passing it the file as a script to interpret.
As written, this script works as an executable script only if `wish` is installed in `/usr/local/bin`, although you could still run it by invoking `wish` with the script file name as a command-line argument. If `wish` has been installed somewhere else, you need to change the first line to reflect its location on

your system. Some systems misbehave in confusing ways if the first line of the script file is longer than 32 characters, so beware if the full path name of the `wish` binary is longer than 27 characters.
To work around these limitations, a common technique for scripts on Unix has been to start script files with the following three lines:

```
#!/bin/sh
# Tcl ignores the next line but 'sh' doesn't \
exec wish "$0" "$@"
```

or the more arcane but more robust version:

```
#!/bin/sh
# Tcl ignores the next line but 'sh' doesn't \
exec wish "$0" ${1+"$@"}
```

In most modern Unix implementations, though, the following will work correctly, as long as `wish` appears in one of the directories in your `PATH` environment variable:

```
#!/usr/bin/env wish
```

1.3.2 Executable Scripts on Windows

On Windows, you can use the standard system tools to associate the `wish` interpreter with a file extension (`.tcl` by convention) so that double-clicking on the icon for a Tcl/Tk script automatically invokes the `wish` interpreter, passing it the name of the file as a script to interpret. Most Windows installers for Tcl/Tk automatically create this association for you. `wish` is typically selected as the default association because most Windows-based Tcl/Tk programs are GUI-based. However, if the majority of your Tcl scripts don't use Tk commands, you could change the default association to invoke `tclsh`.
If you plan to distribute your scripts on multiple platforms, you should include the appropriate `#!` header as discussed in the previous section for Unix executable scripts so that they can be directly executable on Unix systems. On the other hand, Windows doesn't follow the `#!` convention, and the `#!` line is treated as a comment by the `wish` interpreter, so the net effect is that the line is ignored when the script is run on a Windows system.

1.3.3 Executing Scripts in an Interactive Interpreter

In practice, users of Tk applications rarely type Tcl commands; they interact with the applications using the mouse and keyboard in the usual ways you would expect for graphical applications. Tcl works behind the scenes where users don't normally see it. The `hello.tcl` script behaves just the same as an application that has been coded in C with a GUI toolkit and compiled into a binary executable file.
During debugging, though, it is common for application developers to type Tcl commands interactively. For example, you could test the `hello.tcl` script by starting `wish` interactively (type `wish` to your shell instead of `hello.tcl`). Then enter the following Tcl command:

```
source hello.tcl
```

`source` is a Tcl command that takes a file name as an argument. It reads the file and evaluates it as a Tcl script. This generates the same user interface as if you had invoked `hello.tcl` directly from your shell, but you can now enter Tcl commands interactively, too. For example, you could edit the script file to change the `-command` option to

```
-command "puts Good-bye!; exit"
```

then enter the following commands interactively to `wish` without restarting the program:

```
destroy .b
source hello.tcl
```

The first command deletes the existing button, and the second command re-creates the button with the new `-command` option. Now when you click on the button, the `puts` command prints a message on standard output before `wish` exits.

1.4 Variables and Substitutions

Tcl allows you to store values in variables and use those values in commands. For example, consider the following script, which you could enter in either `tclsh` or `wish`:

```
    set a 44
⇒ 44
```

```
    expr $a*4
⇒ 176
```

The first command assigns the value `44` to the variable `a` and returns the variable's value. In the second command, the `$` causes Tcl to perform *variable substitution*: the Tcl interpreter replaces the dollar sign and the variable name following it with the value of the variable, so that the actual argument received by `expr` is `44*4`. Variables need not be declared in Tcl; they are created automatically when set. Variable values can always be represented as strings but may be maintained in a native binary format. Strings may contain binary data and may be of any length. Of course, in this example an error occurs in `expr` if the value of `a` doesn't make sense as an integer or real number.
Tcl also provides *command substitution*, which allows you to use the result of one command in an argument to another command:

```
    set a 44
   set b [expr $a*4]
⇒ 176
```

Square brackets invoke command substitution: everything inside the brackets is evaluated as a separate Tcl script, and the result of that script is substituted into the word in place of the bracketed command. In this example the second argument of the second `set` command is `176`.
The final form of substitution in Tcl is *backslash substitution*, which either adds special meaning to a normal character or takes it away from a special character, as in the following examples:

```
set x \$a
set newline \n
```

The first command sets the variable `x` to the string `$a` (the characters `\$` are replaced with a dollar sign and no variable substitution occurs). The second command sets the variable `newline` to hold a string consisting of the newline character (the characters `\n` are replaced with a newline character).

1.5 Control Structures

The next example uses variables and substitutions along with some simple control structures to create a Tcl *procedure* called `factorial`, which computes the factorial of a given non-negative integer value:

```
proc factorial {val} {
    set result 1
    while {$val>0} {
        set result [expr $result*$val]
        incr val -1
    }
    return $result
}
```

If you enter the preceding lines in `wish` or `tclsh`, or if you enter them into a file and then `source` the file, a new command `factorial` becomes available. The command takes one non-negative integer argument, and its result is the factorial of that number:

```
  factorial 3
⇒ 6
  factorial 20
⇒ 2432902008176640000
  factorial 0.5
∅ expected integer but got "0.5"
```

This example uses one additional piece of Tcl syntax: braces. Braces are like double quotes in that they can be placed around a word that contains embedded spaces. However, braces are different from double quotes in two respects. First, braces nest. The last word of the `proc` command starts after the open brace on the first line and contains everything up to the close brace on the last line. The Tcl interpreter removes the outer braces and passes everything between them, including several nested pairs of braces, to `proc` as an argument. The second difference between braces and double quotes is that no substitutions occur inside braces, whereas they do inside quotes. All of the characters between the braces are passed verbatim to `proc` without any special processing.

The `proc` command takes three arguments: the name of a procedure, a list of argument names separated by whitespace, and the body of the procedure, which is a Tcl script. `proc` enters the procedure name into the Tcl interpreter as a new command. Whenever the command is invoked, the body of the procedure is evaluated. While the procedure body is executing, it can access its arguments as variables: `val` holds the first and only argument.

The body of the `factorial` procedure contains three Tcl commands: `set`, `while`, and `return`. The `while` command does most of the work of the procedure. It takes two arguments, an expression, `$val>0`, and a body, which is another Tcl script. The `while` command evaluates its expression argument and if the result is nonzero, it evaluates the body as a Tcl script. It repeats this process over and over until eventually the expression evaluates to zero. In the example, the body of the `while` command multiplies the result by `val` and then uses the `incr` command to add the specified integer increment (`-1` in this case) to the value contained in `val`. When `val` reaches zero, `result` contains the desired factorial.

The `return` command causes the procedure to exit with the value of the variable `result` as the procedure's result. If a `return` command is omitted, the return value of a procedure is the result of the last command executed in the procedure's body. In the case of `factorial` this would be the result of `while`, which is always an empty string.

The use of braces in this example is crucial. The single most difficult issue in writing Tcl scripts is managing substitutions: making them happen when you want them and preventing them when you don't. The body of the procedure must be enclosed in braces because we don't want variable and command substitutions to occur at the time the body is passed to `proc` as an argument; we want the substitutions to occur later, when the body is evaluated as a Tcl script. The body of the `while` command is enclosed in braces for the same reason: rather than performing the substitutions once, while parsing the `while` command, we want the substitutions to be performed over and over, each time the body is evaluated. Braces are also needed in the `{$val>0}` argument to `while`. Without them the value of the variable `val` would be substituted when the `while` command is parsed; the expression would have a constant value and `while` would loop forever. Try replacing some of the braces in the example with double quotes to see what happens.

The examples in this book use a style in which the open brace for an argument that is a Tcl script appears at the end of one line, the script follows on successive indented lines, and the close brace is on a line by itself after the script. Although this makes for readable scripts, Tcl doesn't require this particular syntax. Arguments that are scripts are subject to the same syntax rules as any other arguments; in fact, the Tcl interpreter doesn't even know that an argument is a script at the time it parses it. One consequence is that the open brace must be on the same line as the preceding portion of the command. If the open brace is moved to a line by itself, the newline before the open brace terminates the command.

The variables in a procedure are normally local to that procedure and are not visible outside the procedure. In the `factorial` example the local variables include the argument `val` as well as the variable `result`. A fresh set of local variables is created for each call to a procedure (arguments are passed by copying their values), and when a procedure returns, its local variables are deleted. Variables named outside any procedure are called *global variables*; they last forever unless explicitly deleted. You'll find out later how a procedure can access global variables and the local variables of other active procedures. Additionally, persistent variables can be created within specific *namespaces* to prevent naming conflicts; Chapter 10 discusses the use of namespaces.

1.6 On the Tcl Language

As a programming language, Tcl is defined quite differently from most other languages. Most languages have a grammar that defines the entire language. For example, consider the following statement in C:

```
while (val>0) {
    result *= val;
    val -= 1;
}
```

The grammar for C defines the structure of this statement in terms of a reserved word `while`, an expression, and a substatement to execute repeatedly until the expression evaluates to zero. The C grammar defines both the overall structure of the `while` statement and the internal structure of its expression and substatement.

In Tcl no fixed grammar explains the entire language. Instead, Tcl is defined by an interpreter that parses single Tcl commands, plus a collection of procedures that execute individual commands. The interpreter and its substitution rules are fixed, but new commands can be defined at any time and existing commands can be replaced. Features such as control flow, procedures, and expressions are implemented as commands; they are not understood directly by the Tcl interpreter. For example, consider the Tcl command that is equivalent to the preceding `while` loop:

```
while {$val>0} {
    set result [expr $result*$val]
    incr val -1
}
```

When this command is evaluated, the Tcl interpreter knows nothing about the command except that it has three words, the first of which is a command name. The Tcl interpreter has no idea that the first argument to `while` is an expression and the second is a Tcl script. Once the command has been parsed, the Tcl interpreter passes the words of the command to `while`, which treats its first argument as an expression and the second as a Tcl script. If the

expression evaluates to nonzero, `while` passes its second argument back to the Tcl interpreter for evaluation. At this point the interpreter treats the contents of the argument as a script (i.e., it performs command and variable substitutions and invokes the `expr`, `set`, and `incr` commands).
Now consider the following command:

```
set {$val>0} {
        set result [expr $result*$val]
        incr val -1
}
```

As far as the Tcl interpreter is concerned, the `set` command is identical to the `while` command except that it has a different command name. The interpreter handles this command in exactly the same way as the `while` command, except that it invokes a different procedure to execute the command. The `set` command treats its first argument as a variable name and its second argument as a new value for that variable, so it will set a variable with the rather unusual name of `$val>0`.
The most common mistake made by new Tcl users is to try to understand Tcl scripts in terms of a grammar; this leads people to expect much more sophisticated behavior from the interpreter than actually exists. For example, a C programmer using Tcl for the first time might think that the first pair of braces in the `while` command serves a different purpose from the second pair. In reality, there is no difference. In each case the braces are present so that the Tcl interpreter passes the characters between the braces to the command without performing any substitutions.
Thus the entire Tcl "language" consists of about a dozen simple rules for parsing arguments and performing substitutions. The actual behavior of a Tcl script is determined by the commands executed. The commands determine whether to treat an argument as a literal value, the name of a variable, a code block to execute, and so on. An interesting consequence of this is that a script can define commands implementing entirely new control structures, which is a feature not available in most other languages.

1.7 Event Bindings

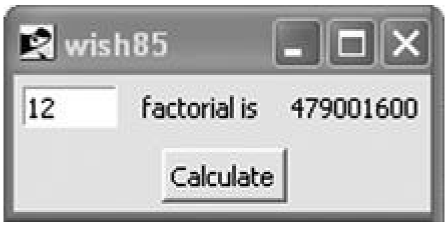
The next example provides a graphical front end for the `factorial` procedure. In addition to demonstrating two new widget classes, it illustrates Tk's *binding* mechanism. A binding causes a particular Tcl script to be evaluated whenever a particular event occurs in a particular window. The `-command`

option for buttons is an example of a simple binding implemented by a particular widget class. Tk also includes a more general mechanism that can be used to extend the behavior of widgets in nearly arbitrary ways.
To run the example, copy the following script into a file `factorial.tcl` and invoke the file from your shell.

```
#!/usr/bin/env wish
proc factorial {val} {
    set result 1
    while {$val>0} {
        set result [expr $result*$val]
        incr val -1
    }
    return $result
}
entry .value -width 6 -relief sunken -textvariable value
label .description -text "factorial is"
label .result -textvariable result
button .calculate -text "Calculate" \
    -command {set result [factorial $value]}
bind .value <Return> {
    .calculate flash
    .calculate invoke
}
grid .value .description .result -padx 1m -pady 1m
grid .calculate - - -padx 1m -pady 1m
```

This script produces a screen display like that in Figure 1.2. There is an entry widget in which you can click with the mouse and type a number. If you click the button labeled “Calculate,” the result appears on the right side of the window; the same occurs if you press the Return key in the entry.

Figure 1.2 A graphical user interface that computes a factorial



This application consists of four widgets: one entry, one button, and two labels. Entries are widgets that display one-line text strings that you can edit interactively. The entry is configured with a `-width` of `6`, which means it is large enough to display about six digits, and a `-relief` of `sunken`, which makes the entry appear sunken into the window. The `-textvariable` option for each entry specifies the name of a global variable to hold the entry’s text—any

changes you make in the entry are reflected in the variable and vice versa.
The `.description` label widget holds decorative text, and the `.result` label holds the result of the power computation. The `-textvariable` option for `.result` causes it to display whatever string is in the global variable `result` and to update itself whenever the variable changes. In contrast, `.description` displays a constant string.
The first `grid` command arranges the entry and two label widgets in a row from left to right. The `-padx` and `-pady` options make the display a bit more attractive by arranging for 1 millimeter of extra space on the left and right sides of each widget, and 1 millimeter of extra space above and below each widget. The `m` suffix specifies millimeters; you could also use `c` for centimeters, `i` for inches, `p` for points, or no suffix for pixels.
The second `grid` command arranges the button in a second row. Because the widget name occurs as the first argument, the gridder allocates the first column of the row to the button. The two `-` arguments following the widget name indicate to the gridder that the space allocated to the button widget should span two additional columns. The gridder then centers the button widget inside its allocated space.
The command creating the `.calculate` button occupies two lines in the script; the backslash at the end of the first line is a line-continuation character, which causes the newline to be treated as a space. The button's `-command` script connects the user interface to the `factorial` procedure. The script invokes `factorial`, passing it the values in the entry and storing the result in the `result` variable so that it is displayed in the `.result` widget.
The `bind` command has three arguments: the name of a widget, an event specification, and a Tcl script to invoke when the given event occurs in the given widget. `<Return>` specifies an event consisting of the user pressing the return key on the keyboard (which is still labeled "Return" on Mac keyboards but typically labeled "Enter" on most other English keyboards these days). Table 1.1 shows a few other event specifiers that you might find useful.

Table 1.1 Event Specifiers

Event specifer	Meaning
`<Button-1>`	Mouse button 1 is pressed
`<1>`	Shorthand for `<Button-1>`
`<ButtonRelease-1>`	Mouse button 1 is released
`<Double-Button-1>`	Double-click on mouse button 1
`<Key-a>`	Key a is pressed
`<a>` or a	Shorthand for `<Key-a>`
`<Motion>`	Pointer motion with any (or no) buttons or modifier keys pressed
`<B1-Motion>`	Pointer motion with button 1 pressed

The script for a binding has access to several pieces of information about the event, such as the location of the pointer when the event occurred. For an example, start up `wish` interactively and enter the following command in it:

```
bind . <Motion> {puts "pointer at %x,%y"}
```

Now move the pointer over the window. Each time the pointer moves, a message is printed on standard output giving its new location. When the pointer motion event occurs, Tk scans the script for `%` sequences and replaces them with information about the event before passing the script to Tcl for evaluation. `%x` is replaced with the pointer's x-coordinate and `%y` is replaced with the pointer's y-coordinate.
The intent of a binding is to extend the generic built-in behavior of the entry (editing text strings) with an application-specific behavior. In this script, as a convenience we would like to allow the user to request the factorial calculation by pressing the Return key as an alternative to clicking the "Calculate" button. We could simply duplicate the button's command script, but if we were to modify the command script later, we'd need to remember to replicate the change in the binding script as well. Instead, we provide a binding script that "programmatically clicks" the button.
The binding script executes two commands called *widget commands*. Whenever a new widget is created, a new Tcl command is also created with the same name as the widget, and you can invoke this command to communicate with the widget. The first argument to a widget command selects one of several operations, and additional arguments are used as parameters for that operation. In this binding script, the first widget command flashes the button. (Depending on your system's color scheme, you might not see the button flash.) The second widget command causes the button widget to invoke its `-command` option just as if you had clicked the mouse button on it.

Each class of widget supports a different set of operations in its widget commands, but many of the operations are similar from class to class. For example, every widget class supports a `configure` widget command that can be used to modify any of the configuration options for the widget. If you run the `factorial.tcl` script interactively, you could type the following command to change the background of the entry widget to yellow:

```
.value configure -background yellow
```

Or you could type

```
.calculate configure -state disabled
```

to make the button unresponsive to user interaction.

1.8 Additional Features of Tcl and Tk

The examples in this chapter have used almost every aspect of the Tcl language syntax, and they illustrated many features of Tcl and Tk. However, Tcl and Tk contain many other facilities that are not used in this chapter; all of these are described later in the book. Here is a sample of some of the most useful features that haven't been mentioned yet:

- **Arrays, dictionaries, and lists**—Tcl provides associative arrays and dictionaries for storing key-value pairs efficiently and lists for managing aggregates of data.
- **More control structures**—Tcl provides several additional commands for controlling the flow of execution, such as `eval`, `for`, `foreach`, and `switch`.
- **String manipulation**—Tcl contains a number of commands for manipulating strings, such as measuring their length, regular expression pattern matching and substitution, and format conversion.
- **File access**—You can read and write files from Tcl scripts and retrieve directory information and file attributes such as size and creation time.
- **More widgets**—Tk contains many widget classes besides those shown here, such as menus, scrollbars, a drawing widget called a *canvas*, and a text widget that makes it easy to achieve hypertext effects.
- **Access to other windowing features**—Tk provides commands for

accessing all of the major windowing facilities, such as a command for communicating with the window manager (to set the window's title, for example), a command for retrieving the selection, and a command to manage the input focus.

- **Interapplication communication**—Tcl includes the ability to communicate between applications through interprocess pipes and TCP/IP sockets.
- **C interfaces**—Tcl provides C library procedures that you can use to define new Tcl commands in C. (Tk provides a library that you can use to create new widget classes and geometry managers in C, but this capability is rarely used and so is not covered in this book.)

2. Tcl Language Syntax

To write Tcl scripts, you must learn two things. First, you must learn the Tcl syntax, which consists of a dozen rules that determine how commands are parsed. The Tcl syntax is the same for every command. Second, you must learn about the individual commands that you use in your scripts. Tcl provides about 100 built-in commands, Tk adds several dozen more, and any application based on Tcl or Tk will add a few more of its own. You'll need to know all of the syntax rules right away, but you can learn about the commands more gradually as you need them.
This chapter describes the Tcl language syntax. The remaining chapters in Part I describe the built-in Tcl commands, and Part II describes Tk's commands.

2.1 Scripts, Commands, and Words

A Tcl *script* consists of one or more *commands*. Commands are separated by newlines or semicolons. For example,

```
set a 24
set b 15
```

is a script with two commands separated by a newline character. The same script could be written on a single line using a semicolon separator:

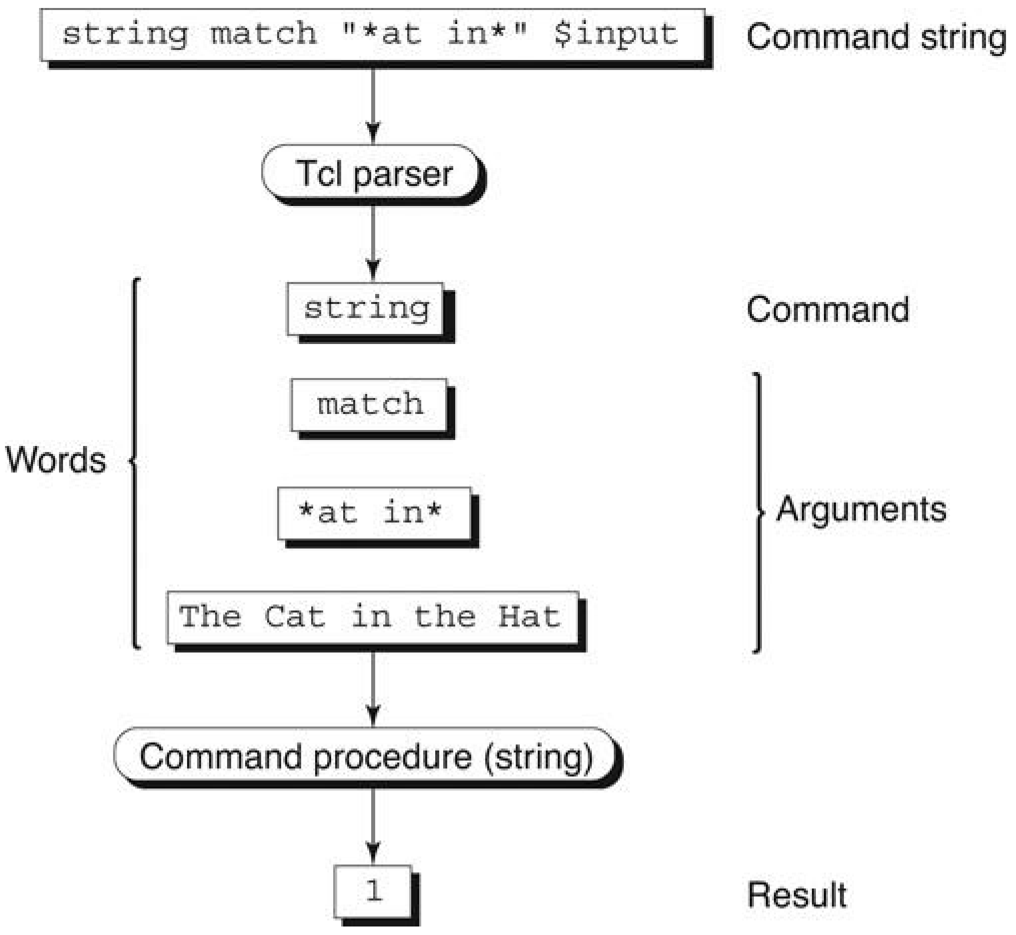
```
set a 24; set b 15
```

Each command consists of one or more *words*, where the first word is the name of a command and additional words are arguments to that command. Words are separated by spaces or tabs, which are generally referred to as either *whitespace* characters or just whitespace. Each of the commands in the preceding examples has three words. There may be any number of words in a command, and each word may have an arbitrary string value. The whitespace that separates words is not part of the words, nor are the newlines and semicolons that terminate commands.

2.2 Evaluating a Command

Tcl evaluates a command in two steps as shown in Figure 2.1: *parsing* and *execution*. In the parsing step the Tcl interpreter applies the rules described in this chapter to divide the command up into words and perform substitutions. Parsing is done in exactly the same way for every command. During the parsing step, the Tcl interpreter does not apply any meaning to the values of the words. Tcl just performs a set of simple string operations such as replacing the characters `$input` with the string stored in the variable `input`; Tcl does not know or care whether the resulting word is a number or the name of a widget or anything else.

Figure 2.1 The parsing and execution of Tcl commands



In the execution step, meaning is applied to the words of the command. Tcl treats the first word as a command name, checking to see if the command is defined and locating a *command procedure* to carry out its function. If the command is defined, the Tcl interpreter invokes its command procedure, passing all of the words of the command to the command procedure. The command procedure assigns meaning to these words according to its own needs; because each command does something different, each command applies different meaning to its arguments.

Note

The Tcl community uses the terms *word* and *argument* interchangeably to refer to the values passed to command procedures. The only difference between these two terms is that the first argument is the second word.

The following commands illustrate some of the meanings that are commonly applied to arguments:

- `set a 122`

In many cases, such as the `set` command, arguments may take any form whatsoever. The `set` command simply treats the first argument as a variable name and the second argument as a value for the variable. The command `set 122 a` is valid, too: it creates a variable whose name is `122` and whose value is `a`.

- `expr 24 / 3.2`

The `expr` command concatenates its arguments, and the result must be an arithmetic expression that follows the rules described in Chapter 4. Several other commands also take expressions as arguments.

- `lindex {red green blue purple} 2`

The first argument to `lindex` is a *list* consisting of four values separated by spaces. This command returns the value of the element at index `2` in the list (actually the third element, `blue`, as elements are numbered starting at 0). Tcl's commands for manipulating lists are described in Chapter 6.

- `string length abracadabra`

Some commands, like `string` and the Tk widget commands, are actually several commands rolled into one. The first argument of the command selects one of several operations to perform and determines the meaning of the remaining arguments. For example, `string length` requires one additional argument and computes its length, whereas `string compare` requires two additional arguments. Such a collection of commands is sometimes referred to as an *ensemble*.

- `button .b -text Hello -fg red`

The arguments after `.b` are option-value pairs that allow you to specify the options you care about and use default values for the others.

When writing Tcl scripts, one of the most important things to remember is that the Tcl parser doesn't apply any meaning to the words of a command while it parses them. All of the preceding meanings are applied by individual command procedures, not by the Tcl parser. This approach is similar to that of most shell languages but different from most programming languages. For example, consider the following C program code:

```
x = 4;
y = x+10;
```

In the first statement C stores the integer value `4` in variable `x`. In the second statement C evaluates the expression `x+10`, fetching the value of variable `x` and adding `10`, and stores the result in variable `y`. At the end of execution, `y` has the integer value `14`. If you want to use a literal string in C without evaluation, you must enclose it in quotes. Now consider a similar-looking program written in Tcl:

```
set x 4
set y x+10
```

The first command assigns the *string* `4` to variable `x`. The value of the variable need not have any particular form. The second command simply takes the string `x+10` and stores it as the new value for `y`. At the end of the script, `y` has the string value `x+10`, not the integer value `14`. In Tcl, if you want evaluation you must ask for it explicitly:

```
set x 4
set y [expr $x+10]
```

Evaluation is requested twice in the second command. First, the second word of the command is enclosed in brackets, which tells the Tcl parser to evaluate the characters between the brackets as a Tcl script and use the result as the value of the word. Second, a dollar sign has been placed before `x`. When Tcl parses the `expr` command, it substitutes the value of variable `x` for the `$x`. If the dollar sign were omitted, `expr`'s argument would contain the string `x`, resulting in a syntax error. At the end of the script, `y` has the string value `14`.

2.3 Variable Substitution

Tcl provides three forms of *substitution*: variables, commands, and backslashes. Each substitution causes some of the original characters of a word to be replaced with some other value. The Tcl interpreter performs the substitutions before executing the command procedure. Substitutions may occur in any word of a command, including the command name itself, and there may be any number of substitutions within a single word.

Note

Substitutions do not affect the word boundaries of a command, even if the characters substituted contain whitespace characters such as spaces, tabs, or newlines.

The first form of substitution is *variable substitution*. It is triggered by a dollar sign character, and it causes the value of a Tcl variable to be inserted into a word. For example, consider the following commands:

```
   set kgrams 20
   expr $kgrams*2.2046
⇒ 44.092
```

The first command sets the value of the variable `kgrams` to `20`. The second command computes the corresponding weight in pounds by multiplying the value of `kgrams` by `2.2046`. It does this using variable substitution: the string `$kgrams` is replaced with the value of the variable `kgrams`, so that the actual argument received by the `expr` command procedure is `20*2.2046`.
Variable substitution can occur anywhere within a word and any number of times, as in the following command:

```
expr $result*$base
```

The variable name consists of all of the numbers, letters, and underscores following the dollar sign. Thus the first variable name (`result`) extends up to the `*` and the second variable name (`base`) extends to the end of the word. Variable substitution can be used for many purposes, such as generating new names:

```
foreach num {1 2 3 4 5} {
     button .b$num
}
```

This example creates five button widgets, with names `.b1`, `.b2`, `.b3`, `.b4`, and `.b5`.

These examples show only the simplest form of variable substitution. Two other forms of variable substitution are used for associative array references and to provide more explicit control over the extent of a variable name (e.g., so that there can be a letter or number immediately following the variable name). These other forms are discussed in Chapter 3.

2.4 Command Substitution

The second form of substitution provided by Tcl is *command substitution*. Command substitution causes part or all of a word to be replaced with the result of a Tcl command. Command substitution is invoked by enclosing a command in brackets:

```
   set kgrams 20
   set lbs [expr $kgrams*2.2046]
⇒ 44.092
```

The characters between the brackets must constitute a valid Tcl script. The script may contain any number of commands separated by newlines or semicolons in the usual fashion. The brackets and all of the characters between them are replaced with the result of the script. Thus in the foregoing example the `expr` command is executed while the words for `set` are parsed; its result, the string `44.092`, becomes the second argument to `set`. As with variable substitution, command substitution can occur anywhere in a word, and there may be more than one command substitution within a single word.

2.5 Backslash Substitution

The final form of substitution in Tcl is *backslash substitution*. It is used to insert special characters such as newlines into words and also to insert characters such as `[` and `$` without their being treated specially by the Tcl parser. For example, consider the following command:

```
   set msg Eggs:\ \$2.18/dozen\nGasoline:\ \$2.49/gallon
⇒ Eggs: $2.18/dozen
   Gasoline: $2.49/gallon
```

There are two sequences of a backslash followed by a space; each of these sequences is replaced in the word by a single space, and the space characters are not treated as word separators. There are also two sequences of a backslash followed by a dollar sign; each of these is replaced in the word with a single dollar sign, and the dollar signs are treated like ordinary characters (they do not trigger variable substitution). The backslash followed by `n` is replaced with a newline character.

Table 2.1 lists all of the backslash sequences supported by Tcl. These include all of the sequences defined for ANSI C, such as `\t` to insert a tab character and `\x7d` to insert the character whose hexadecimal value is 0x007d in the Unicode character encoding. (Chapter 5 describes the Unicode encoding in greater depth.) If a backslash is followed by any character not listed in the table, as in `\$` or `\[`, the backslash is dropped from the word and the following character is included in the word as an ordinary character. This allows you to include any of the Tcl special characters in a word without the characters being treated specially by the Tcl parser. The sequence `\\` inserts a single backslash into a word. And a backslash followed by a space character inserts a space as a literal character, rather than treating it as word-delimiting whitespace.

Table 2.1 Backslash Substitutions Supported by Tcl

Backslash sequence	**Replaced by**
`\a`	Audible alert (0x7)
`\b`	Backspace (0x8)
`\f`	Form feed (0xc)
`\n`	Newline (0xa)
`\r`	Carriage return (0xd)
`\t`	Tab (0x9)
`\v`	Vertical tab (0xb)
`\`*ooo*	Unicode character specified by 8-bit octal value *ooo* (one, two, or three *o*'s)
`\x`*hh*	Unicode character specified by 8-bit hex value *hh* (any number of *h*'s, although all but the last two are ignored)
`\u`*hhhh*	Unicode character specified by 16-bit hex value *hhhh* (from one to four *h*'s)
`\`*newline whitespace*	A single space character

The sequence backslash-newline can be used to spread a long command across multiple lines, as in the following example:

```
pack .base .label1 .power .label2 .result \
        -side left -padx 1m -pady 2m
```

The backslash and newline, plus any leading whitespace on the next line, are replaced by a single space character in the word. Thus the two lines together form a single command.

Note

Backslash-newline sequences are unusual in that they are replaced in a separate preprocessing step before the Tcl interpreter parses the command. This means, for example, that the space character that replaces backslash-newline will be treated as a word separator unless it is between double quotes or braces.

2.6 Quoting with Double Quotes

Tcl provides several ways for you to prevent the parser from giving special interpretation to characters such as `$` and the semicolon. These techniques are called *quoting*. You have already seen one form of quoting in backslash substitution. For example, `\$` causes a dollar sign to be inserted into a word without triggering variable substitution. In addition to backslash substitution, Tcl provides two other forms of quoting: double quotes and braces. Double quotes disable word and command separators, and braces disable all special characters.
If the first character of a word is a double quote, the word is terminated by the next double-quote character. Note that the double quotes themselves are not part of the word; they are simply delimiters. If a word is enclosed in double quotes, then spaces, tabs, newlines, and semicolons are treated as ordinary characters within the word. The example from the previous section can be rewritten more cleanly with double quotes as follows:

```
  set msg "Eggs: \$2.18/dozen\nGasoline: \$1.49/gallon"
⇒ Eggs: $2.18/dozen
  Gasoline: $1.49/gallon
```

The `\n` in the example could also be replaced with an actual newline character, as in

```
set msg "Eggs: \$2.18/dozen
Gasoline: \$1.49/gallon"
```

but many consider the script more readable with `\n`.
If the word does not start with a double-quote character, any double-quote character within the word is treated as a literal character, not a delimiter. For example, the following is a legal, but highly discouraged, usage of literal double-quote characters in Tcl:

```
  puts This"is"poor"usage
⇒ This"is"poor"usage
```

Variable substitutions, command substitutions, and backslash substitutions all occur as usual inside double quotes. For example, the following script sets `msg` to a string containing the name of a variable, its value, and the square of its value:

```
  set a 2.1
  set msg "a is $a; the square of a is [expr $a*$a]"
⇒ a is 2.1; the square of a is 4.41
```

If you would like to include a double quote in a word enclosed in double quotes, use backlash substitution:

```
  set name a.out
  set msg "Couldn't open file \"$name\""
⇒ Couldn't open file "a.out"
```

2.7 Quoting with Braces

Braces provide a more radical form of quoting where all the special characters lose their meaning. If a word starts with an open brace, all the characters between it and the matching close brace are the value of the word, verbatim. No substitutions are performed on the word, and spaces, tabs, newlines, and semicolons are treated as ordinary characters. The example from Section 2.5 can be rewritten with braces as follows:

```
set msg {Eggs: $2.18/dozen
Gasoline: $1.49/gallon}
```

The dollar signs in the word do not trigger variable substitution, and the newline does not act as a command separator. In this case `\n` cannot be used to insert a newline into the word, because the `\n` would be included in the argument as is without triggering backslash substitution:

```
  set msg {Eggs: $2.18/dozen\nGasoline: $1.49/gallon}
⇒ Eggs: $2.18/dozen\nGasoline: $1.49/gallon
```

Note

The only form of substitution that occurs between braces is for backslash-newline. As discussed in Section 2.5, backslash-newline sequences are actually removed in a preprocessing step before the command is parsed.

Unlike double quotes, braces can be nested. This is very common for procedure definitions and control flow commands, where one or more of the arguments are scripts to evaluate. Because the script must appear as a single argument, it must be quoted. In turn the script argument might contain other control flow commands with their own script arguments.

Note

If a brace is backslashed, it does not count in finding the matching close brace for a word enclosed in braces. The backslash is not removed when the word is parsed.

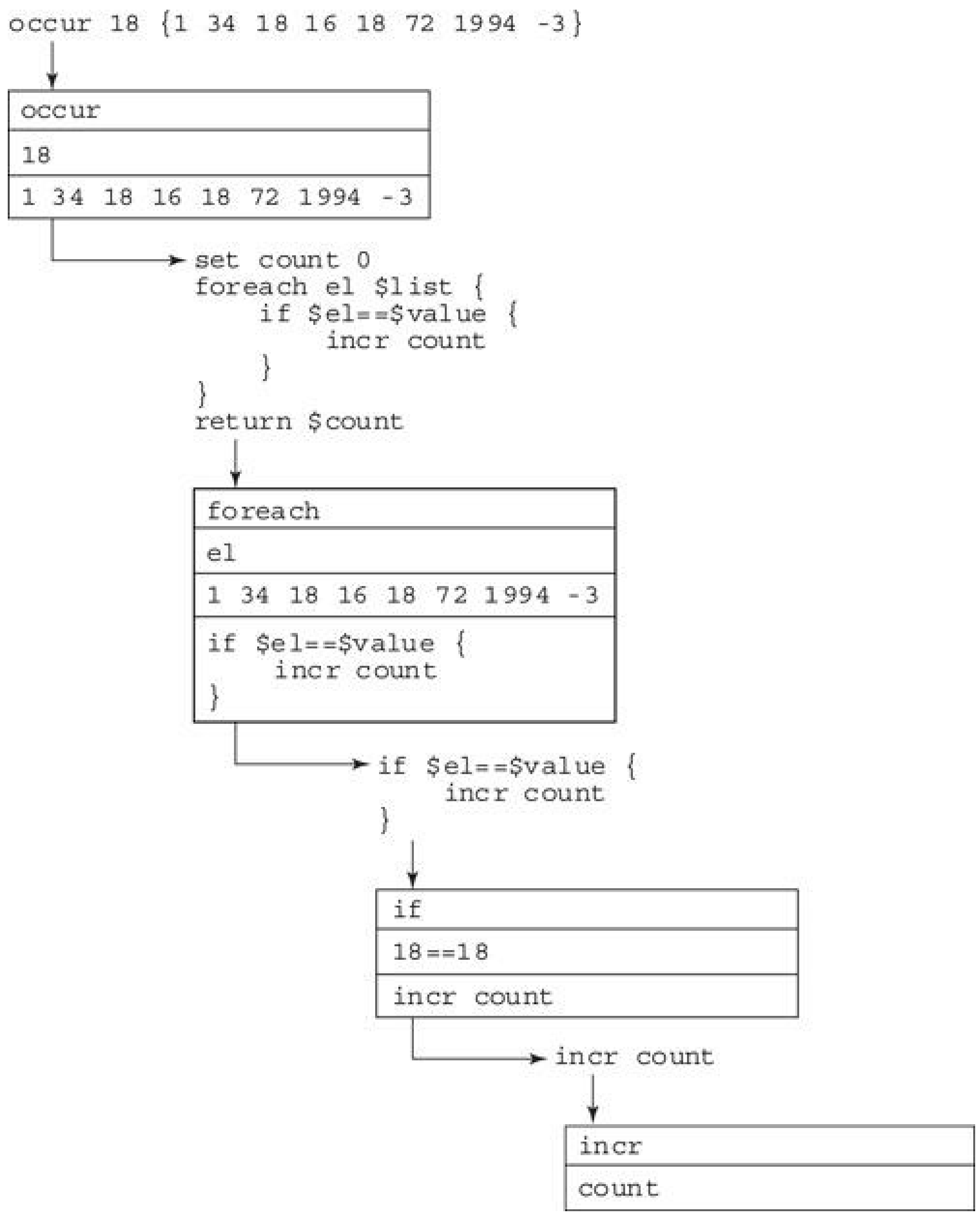
One of the most important uses for braces is to *defer evaluation*. Deferred evaluation means that special characters aren't processed immediately by the Tcl parser. Instead they are passed to the command procedure as part of its argument. The command procedure then processes the special characters itself, often by passing the argument back to the Tcl interpreter for evaluation. For example, consider the following procedure, which counts the number of occurrences of a particular value in a list:

```
proc occur {value list} {
    set count 0
    foreach el $list {
        if $el==$value {
            incr count
        }
    }
    return $count
}
```

The body of the procedure is enclosed in braces so that it is passed verbatim to `proc`. Thus the value of the variable `list` is not substituted at the time the `proc` command is parsed. This is necessary if the procedure is to work correctly: a different value must be substituted for `$list` each time the procedure is invoked. Note that braces nest, so that the last argument to `proc` extends up to the matching close brace. Figure 2.2 illustrates what happens when the following Tcl script is subsequently evaluated:

```
occur 18 {1 34 18 16 18 72 1994 -3}
```

Figure 2.2 A snapshot of nested scripts being evaluated



The braces around the second argument to `occur` cause the entire list of numbers to be passed to `occur` as a single word. The Tcl procedure mechanism then passes the procedure's body to the Tcl interpreter for evaluation. When Tcl parses the `foreach` command in the body, the value of the variable `list` is substituted, but the last argument to `foreach` (the loop body) is enclosed in braces so no substitutions are performed on it. The `foreach` command procedure sets the variable `el` to each element of the list in turn and calls the Tcl interpreter to evaluate the loop body for each element. As part of this evaluation, Tcl parses the `if` command, substituting the values of the variables `el` and `value`. In the snapshot shown in Figure 2.2, the `if` test succeeded, so `if` evaluates the `incr` command.

2.8 Argument Expansion

The example in Section 2.7 demonstrated passing a list as an argument to a procedure. In that case the list appeared literally on the command line. Commonly, the list value might be the result of a variable or command substitution. But because the Tcl interpreter makes substitutions in a single pass from left to right, the list value is treated as a single word; the spaces embedded in the list value are not regarded as word separators.
In some situations the single-layer-of-substitutions rule can be a hindrance rather than a help. For example, the following script is an erroneous attempt to delete all files with names ending in `.o`:

```
file delete [glob *.o]
```

The `glob` command returns a list of all file names that match the pattern `*.o`, such as `a.o b.o c.o`. However, the entire list of files is passed to `file delete` as a single argument; `file delete` silently fails because it cannot find a file named `a.o b.o c.o`. For `file delete` to work correctly, the result of `glob` must be split into multiple words. You can accomplish this through argument expansion.
If a word starts with the string `{*}` followed by a non-whitespace character, Tcl removes the leading `{*}` and parses and substitutes the rest of the word as it would any other word. After substitution, Tcl parses the word again, but without substitution, to verify that the content consists of one or more syntactically complete words. If this is the case, the words are added individually to the command being evaluated; otherwise, Tcl raises a syntax error. Thus,

```
file delete {*}[glob *.o]
```

would be equivalent to

```
file delete a.o b.o c.o
```

after the Tcl interpreter performs substitution and expansion of the argument.
The `{*}` syntax first appeared in Tcl 8.5. Prior versions require you to add layers of parsing explicitly if you want them. Remember that Tcl commands are evaluated in two phases: parsing and execution. The substitution rules apply only to the parsing phase. Once Tcl passes the words of a command to a command procedure for execution, the command procedure can do

anything it likes with them. Some commands reparse their words—for example, by passing them back to the Tcl interpreter. `eval` is an example of such a command, and it provides an alternate solution to the problems with `file delete`:

```
eval file delete [glob *.o]
```

`eval` concatenates all of its arguments with spaces in between and then evaluates the result as a Tcl script, at which point another round of parsing and evaluation occurs. In this example `eval` receives three arguments: `file`, `delete`, and `a.o b.o c.o`. It concatenates them to form the string `file delete a.o b.o c.o`. When this string is parsed as a Tcl script, it yields five words. Each of the file names is passed to `file delete` as a separate argument, so the files are all removed successfully. See Section 8.6 for more details on the proper use of the `eval` command.

2.9 Comments

If the first nonblank character of a command is `#`, the `#` and all the characters following it up through the next newline are treated as a comment and discarded. Note that the hashmark must occur in a position where Tcl is expecting the first character of a command. If a hashmark occurs anywhere else, it is treated as an ordinary character that forms part of a command word:

```
  # This is a comment
  set a 100               # Not a comment
∅ wrong # args: should be "set varName ?newValue?"
  set b 101               ;# This is a comment
⇒ 101
```

The `#` on the second line is not treated as a comment character because it occurs in the middle of a command. As a result, the first `set` command receives six arguments and generates an error. The last `#` is treated as a comment character, since it occurs just after a command was terminated with a semicolon.
Tcl’s simple, consistent syntax rules can have some unexpected consequences. The requirement that the hashmark starting a comment must occur in a position where Tcl is expecting a command implies that the following example is not a comment:

```
set example {
    # This is not a comment
}
```

Instead, all of the characters occurring between the braces are treated as a single argument and are used as the string value assigned to the variable by the `set` command. In contrast, consider the following example:

```
if {$x < 0} {
    # This is a comment
    puts "The result is negative."
}
```

In this case, the Tcl parser identifies two brace-quoted arguments that it passes to the `if` command. The `if` command evaluates its first argument as a Boolean expression, and if the result is true, it invokes the Tcl interpreter recursively to execute the second argument as a Tcl script. It is during this recursive invocation of the Tcl interpreter that the line beginning with the hashmark is identified as a comment.
Another consequence of Tcl's consistent syntax rules is that braces that appear inside a comment often cause errors when you execute the code. Consider the following example:

```
proc countdown {x} {
    puts "Running countdown"
    # Incorrectly comment out this code block {
        while { $x >= 0 } {
            puts "x = $x"
            incr x -1
        }
    }
}
```

When the Tcl interpreter parses this command, the `{` at the end of the first line marks the beginning of a word. The matching `}` marking the end of the word occurs on the last line. The fact that the word contains a script is irrelevant; Tcl is only parsing the command line into a set of words at this point, and so it follows a simple process of finding the matching brace, taking into account any and all nested braces contained within the word. Tcl then passes the word to the `proc` command, which stores it as the script implementing the `countdown` procedure.
It's when you attempt to execute the `countdown` procedure that Tcl then parses and executes the script. After executing the first `puts` command, Tcl ignores the entire second line—including the brace—as a comment. The `while` command is well formed, with its script argument terminated by the `}`

following the `incr` command. Because Tcl then expects to find another command on the following line, the `}` character at the beginning of the line is interpreted as being the name of the command to execute, resulting in the following error:

```
  countdown 3
⇒ Running countdown
  x = 3
  x = 2
  x = 1
  x = 0
∅ invalid command name "}"
```

Note

In general, try to avoid including brace characters in comments. If you do include braces in comments, make sure that they are balanced; for every opening brace, have a corresponding close brace in the comment.

The following example shows a code block correctly commented out. The braces are balanced within the comments:

```
proc countdown {x} {
    puts "Running countdown"
    # while { $x >= 0 } {
    #    puts "x = $x"
    #    incr x -1
    # }
}
```

A common alternative for commenting out a well-formed, parsable block of code is to enclose it within an `if` command with a false condition:

```
proc countdown {x} {
    puts "Running countdown"
    if 0 {
        while { $x >= 0 } {
            puts "x = $x"
            incr x -1
        }
    }
}
```

2.10 Normal and Exceptional Returns

A Tcl command can complete in several different ways. A *normal return* is the most common case; it means that the command completed successfully and the return includes a string result. Tcl also supports *exceptional returns* from commands. An error is the most frequent form of exceptional return. When an error return occurs, it means that the command could not complete its intended function. The command is aborted and any commands that follow it in the script are skipped. An error return includes a string identifying what went wrong; the string normally is displayed by the application. For example, the following `set` command generates an error because it has too many arguments:

```
 set state West Virginia
∅ wrong # args: should be "set varName ?newValue?"
```

Different commands generate errors under different conditions. For example, `expr` accepts any number of arguments but requires the arguments to have a particular syntax; it generates an error if, for example, parentheses aren't matched:

```
 expr 3 * (20+4
∅ unbalanced open paren
 in expression "3 * (20+4"
```

The complete exceptional return mechanism for Tcl is discussed in Chapter 13. Tcl supports a number of exceptional returns other than errors, provides additional information about errors besides the error message mentioned previously, and allows errors to be "caught" so that the effects of the error can be contained within a piece of Tcl code. For now, though, all you need to know is that commands normally return string results, but they sometimes return errors that cause Tcl command interpretation to be aborted.

Note

You may also find the global `errorInfo` variable useful. After an error Tcl sets `errorInfo` to hold a stack trace indicating exactly where the error occurred. You can print out this variable with the command `puts $errorInfo`.

2.11 More on Substitutions

The most common difficulty for new Tcl users is understanding when substitutions do and do not occur. A typical scenario is for a user to be surprised at the behavior of a script because a substitution didn't occur when the user expected it to happen, or a substitution occurred when it wasn't expected. However, you should find Tcl's substitution mechanism to be simple and predictable if you just remember two related rules:

1. Tcl parses a command and makes substitutions in a single pass from left to right. Each character is scanned exactly once.
2. At most a single layer of substitution occurs for each character; the result of one substitution is not scanned for further substitutions.

Tcl's substitutions are simpler and more regular than you may be used to if you've programmed with Unix shells. When new users run into problems with Tcl substitutions, it is often because they have assumed a more complex model than actually exists.
For example, consider the following command:

```
  set x [format {Earnings for July: $%.2f} $earnings]
⇒ Earnings for July: $1400.26
```

The characters between `[]` are scanned exactly once, during command substitution, and the value of the `earnings` variable is substituted at that time. It is *not* the case that Tcl first scans the whole `set` command to substitute variables, then makes another pass to perform command substitution; everything happens in a single scan. The result of the `format` command is passed verbatim to `set` as its second argument without any additional scanning (for example, the dollar sign in `format`'s result does not trigger variable substitution).
One consequence of the substitution rules is that unless you use the `{*}` syntax described in Section 2.8, all the word boundaries within a command are immediately evident and are not affected by substitutions. For example, consider the following script:

```
set city "Los Angeles"
set bigCity $city
```

The second `set` command is guaranteed to have exactly three words regardless of the value of the variable `city`. In this case `city` contains a space character but the space is *not* treated as a word separator.

One final note: It is possible to use substitutions in very complex ways, but you should avoid doing so whenever possible. Substitutions work best when used in very simple ways such as `set a $b`. If you use too many substitutions in a single command, and particularly if you use many backslashes, your code will be unreadable and unreliable. In situations like these, it's usually clearer and safer to break up the offending command into several commands that build up the arguments in simple stages. Tcl provides several commands, such as `format`, `subst`, and `list`, to help manage complex substitution scenarios. Another approach to consider is creating procedures to isolate complex processing; the result is often more readable and maintainable than trying to do all processing "inline."

3. Variables

Tcl supports two kinds of variables: simple variables (often referred to as *scalar* variables) and associative arrays. This chapter describes the basic Tcl commands for manipulating variables and arrays, and it also provides a more complete description of variable substitution.

3.1 Commands Presented in This Chapter

This chapter discusses the following basic commands for manipulating variables:

- `append` *`varName value`* `?`*`value ...`*`?`

Appends each of the *`value`* arguments to the variable *`varName`*, in order. If *`varName`* doesn't exist, it is created with an empty value before appending occurs. The return value is the new value of *`varName`*.

- `incr` *`varName`* `?`*`increment`*`?`

Adds *`increment`* to the value of the variable *`varName`*. Both *`increment`* and the old value of *`varName`* must be integer strings (decimal, hexadecimal, or octal). If the argument is omitted, *`increment`* defaults to `1`. The new value is stored in *`varName`* as a decimal string and returned as the result of the command. As of Tcl 8.5, incrementing a nonexistent variable creates the variable and sets it to the increment value.

- `set` *`varName`* `?`*`value`*`?`

If *`value`* is specified, sets the value of the variable *`varName`* to *`value`*. In any case the command returns the (new) value of the variable.

- `unset ?-nocomplain? ?--?` *`varName`* `?`*`varName varName ...`*`?`

Deletes the variables given by the *`varName`* arguments. Returns an empty string. Raises an error if any of the variables doesn't exist, unless the `-nocomplain` option is included.

- `array exists` *`arrayName`*

Returns a Boolean value indicating whether the array called *`arrayName`* exists.

- `array get` *`arrayName`* `?`*`pattern`*`?`

Returns a dictionary containing the contents of the array called *`arrayName`*. If *`pattern`* is specified, it restricts what elements are present in the dictionary to those whose keys match the pattern according to the rules of *`string match`*.

- `array names` *`arrayName`* `?`*`mode`*`? ?`*`pattern`*`?`

Returns a list of the names of elements within the array called *arrayName*. If *pattern* is specified, it restricts what element names are returned by forming a glob pattern in the style of `string match`. If *mode* is also given, it should be either `-exact`, `-glob`, or `-regexp` meaning that exact matching (like `string equal`), glob matching (like `string match`, the default matching rule), or regular expression matching (like `regexp`) should be used to interpret *pattern* when selecting the element names.

- `array set` *`arrayName dictionary`*

Merges the contents of *dictionary* into the array called *arrayName*. Returns the empty string.

- `array size` *`arrayName`*

Returns the number of elements in the array called *arrayName*.

- `array statistics` *`arrayName`*

Returns a description of the internal configuration of the array named *arrayName*.

- `array unset` *`arrayName ?pattern?`*

Removes all elements that match *pattern* (according to the rules of `string match`) from the array named *arrayName* and returns the empty string. If *pattern* is absent, the whole array is unset.

3.2 Simple Variables and the `set` Command

A simple Tcl variable has a name and a value. Both the name and the value may be arbitrary strings of characters. For example, it is possible to have a variable named `xyz !# 22` or `March earnings: $100,472`. In practice, variable names usually start with a letter and consist of a combination of letters, digits, and underscores, since that makes it easier to use variable substitution. Variable names, like all names in Tcl, are case-sensitive: `numwords` refers to a different variable than `NumWords`.

Variables may be created, read, and modified with the `set` command, which takes either one or two arguments. The first argument is the name of a variable, and the second, if present, is a new value for the variable:

```
  set a {Four score and seven years ago}
⇒ Four score and seven years ago
  set a
⇒ Four score and seven years ago
  set a 12.6
⇒ 12.6
```

Tcl variables are created automatically when they are assigned values. In this example, the first command creates a new variable `a` if it doesn't already exist and sets its value to the character sequence `Four score and seven years ago`. The result of the command is the new value of the variable. The second `set` command has only one argument: `a`. In this form it simply returns the current value of the variable. The third `set` command changes the value of `a` to `12.6` and returns that new value.

3.3 Tcl's Internal Storage of Data

Tcl variables can be used to represent many things, such as integers, real numbers, names, lists, and Tcl scripts, and many of these types of values have a very efficient internal representation. But all values have a string representation as well. The Tcl interpreter automatically converts the value between its internal and string representations on an as-needed basis. For example, if we create the variable `a` with the following command, Tcl stores only the string representation initially:

```
set a 12.6
```

During execution of the `expr` subcommand in the following command, Tcl parses the string value to generate an internal floating-point representation of the value in addition to the string value:

```
set a [expr $a + 1.2]
```

When the `set` command then assigns the return value of `expr` to `a`, only the floating-point internal representation exists. However, if you then use `puts` to print the value of `a` or otherwise treat the value of `a` as a string, Tcl generates a string representation of the value while maintaining the internal representation:

```
puts $a
```

Finally, if you modify the string representation of `a`, such as using `append` to append characters to the end of the value, Tcl updates the string representation and discards the internal floating-point representation (even if the resulting string could be interpreted as a number):

```
append a 32
```

The internal representation enables efficient handling of the value, and the string representation allows different values to be manipulated in the same way and communicated easily. Additionally, because all values appear to be strings at the scripting level, and the Tcl interpreter automatically manages memory allocation for values, there is no need for variable declarations. But by converting between string and internal representations only as needed, Tcl gains significant performance improvement.

Note

Most Tcl programs benefit from this automatic conversion between a value's internal and string formats. However, if your program frequently changes a value while also executing commands that alternate between the internal and string representations of the value, the conversion overhead can be noticeable. This effect, which is called "shimmering," is most pronounced when manipulating large lists, particularly in tight loops. To minimize shimmering, try to manipulate the value with commands that do not modify the internal representation (e.g., use list commands to manipulate lists, arithmetic commands to manipulate numbers, etc.) and avoid forcing a string representation wherever possible.

3.4 Arrays

In addition to simple variables, Tcl also provides *arrays*. An array is a collection of *elements*, each of which is a variable with its own name and value. The name of an array element has two parts: the name of the array and the name of the element within that array. Both array names and element names may be arbitrary strings. For this reason Tcl arrays are sometimes called *associative arrays* to distinguish them from arrays in other languages where the element names must be integers.
Array elements are referenced using notation like `earnings(January)`, where the array name (`earnings` in this case) is followed by the element name in parentheses (`January` in this case). Arrays may be used wherever simple variables may be used, such as in the `set` command:

```
  set earnings(January) 87966
⇒ 87966
  set earnings(February) 95400
⇒ 95400
  set earnings(January)
⇒ 87966
```

The first command creates an array named `earnings`, if it doesn't already exist. Then it creates an element `January` within the array, if it doesn't already exist, and assigns it the value `87966`. The second command assigns a value to the `February` element of the array, and the third command returns the value of the `January` element.

Note

Don't put quotes around the element name. If you do, the quote characters are interpreted as part of the element name, rather than as delimiting the element name.

Although you can use array element names such as `0`, `1`, `2`, etc., the names are still interpreted as strings rather than integer values. Therefore, an element name of `1` is not the same as `1.0`.
In Tcl, arrays are unordered data structures. (Internally, Tcl stores the array elements in a hash table.) Of course, you can assign element names that you can traverse in a particular order, using commands described later in this chapter. But if your intent is to design an ordered data structure, so that values maintain a given sequence, Tcl lists typically provide a better solution. See Chapter 6 for more information on Tcl lists.

3.5 Variable Substitution

Chapter 2 introduced the use of `$` notation for substituting variable values into Tcl commands. This section describes the mechanism in more detail.
Variable substitution is triggered by the presence of an unescaped `$` character in a Tcl word. The characters following the `$` are treated as a variable name, and the `$` and name are replaced in the word by the value of the variable. Tcl provides three forms of variable substitution. So far you have seen only the simplest form, which is used like this:

```
expr $a+2
```

In this form the `$` is followed by a variable name consisting of letters, digits, and underscores. The first character that is not a letter, digit, or underscore (`+` in the example) terminates the name. If the character immediately following a `$` is not a letter, digit, or underscore, the `$` is treated as a literal character.
The second form of variable substitution allows array elements to be substituted. This form is like the first one except that the variable name is followed immediately by an element name enclosed in parentheses. Variable, command, and backslash substitutions are performed on the element name in the same way as a command word in double quotes. For example, consider the following script:

```
set yearTotal 0
foreach month {Jan Feb Mar Apr May Jun Jul Aug Sep \
        Oct Nov Dec} {
    set yearTotal [expr $yearTotal+$earnings($month)]
}
```

In the `expr` command, `$earnings($month)` is replaced with the value of an element of the array `earnings`. The element's name is given by the value of the `month` variable, which varies from iteration to iteration.
An element name may contain whitespace characters, but the parentheses are not Tcl quoting characters, and so any unescaped whitespace characters are treated as word separators:

```
  set capital(New Jersey) Trenton
∅ wrong # args: should be "set varName ?newValue?"
  set capital("New Jersey") Trenton
∅ wrong # args: should be "set varName ?newValue?"
```

The second example results in an error because the quote character doesn't appear as the first character of the word, and so Tcl treats it as a literal character rather than a word delimiter. As a result, `capital("New` is one word and `Jersey")` another, and the `set` command receives too many arguments.
To include a whitespace character in an element name, either escape the whitespace character or quote the entire variable reference:

```
  set capital(New\ Jersey) Trenton
⇒ Trenton
  set "capital(South Dakota)" Pierre
⇒ Pierre
```

Another option is to use substitution for the element name. This works because substitutions do not affect the word boundaries of a command, even if the characters substituted contain whitespace characters; for example:

```
  set state "New Mexico"
⇒ New Mexico
  set capital($state) "Santa Fe"
⇒ Santa Fe
```

The last form of substitution is used for simple variables in places where the variable name is followed by a letter or number or underscore. For example, suppose that you wish to pass a value such as `1.5m` to a command as an argument, but the number is in a variable `size` (in Tk you might do this to specify a size in millimeters). If you try to substitute the variable value with a form like `$sizem`, Tcl will treat the `m` as part of the variable name. To get around this problem, you can enclose the variable name in braces as in the following command:

```
.canvas configure -width ${size}m
```

You can also use braces to specify variable names containing characters other than letters or numbers or underscores.

Note

Braces can be used to delimit only simple variables; there is no way to specify an element of an array with brace notation.

Tcl’s variable substitution mechanism is intended to handle only the most common situations; there exist scenarios where none of the preceding forms of substitution achieves the desired effect. More complicated situations can be handled with a sequence of commands. For example, the `format` command can be used to generate a variable name of almost any imaginable form, `set` can be used to read or write the variable with that name, and command substitution can be used to substitute the value of the variable into other commands.

Note

In general, restrict variable names to sequences of letters, digits, and underscore characters to make your code easier to write and read.

3.6 Multidimensional Arrays

Tcl implements only one-dimensional arrays, but multidimensional arrays can be simulated by concatenating multiple indices into a single element name. The following program simulates a two-dimensional array indexed with integers:

```
    set matrix(1,1) 140
    set matrix(1,2) 218
    set matrix(1,3) 84
    set i 1
    set j 2
    set cell $matrix($i,$j)
⇒ 218
```

`matrix` is an array with three elements whose names are `1,1` and `1,2` and `1,3`. However, the array behaves just as if it were a two-dimensional array; in particular, variable substitution occurs while the element name is scanned in the `set` command, so that the values of `i` and `j` get combined into an appropriate element name.

Note

Spaces are significant in this example: `matrix(1,1)` refers to a different variable than `matrix(1, 1)`. The best practice is to leave out the spaces, since they can also cause confusion during the parsing of commands. For example, the command

```
set matrix(1, 1) 140
```

is a command with four words, of which the third is `1)`. This results in three arguments to `set`, which then causes an error. The entire variable name would have to be enclosed in double quotes to get around this problem.

3.7 Querying the Elements of an Array

The `array` command provides information about the elements currently defined for an array variable, as well as selected operations upon the whole array. It provides this information in several different ways, depending on the first argument passed to it. The command `array size` returns a number indicating how many elements are defined for a given array variable, and the command `array names` returns a list whose entries are the names of the elements of a given array variable:

```
  set currency(France) euro
  set "currency(Great Britain)" pound
  set currency(Japan) yen
  array size currency
⇒ 3
  array names currency
⇒ {Great Britain} Japan France
```

For each of these commands, the second argument must be the name of an array variable. Optionally, you can provide an additional *pattern* argument, in which case the return value contains only those element names that match the pattern using the matching rules of the `string match` command (as described in Section 5.10). The list returned by `array names` does not have any particular order.
The `array names` command can be used in conjunction with `foreach` to iterate through the elements of an array. For example, this code deletes all elements of an array with values that are `0` or empty:

```
foreach i [array names a] {
    if {($a($i) == "") || ($a($i) == 0)} {
        unset a($i)
    }
}
```

Note

The `array` command also provides a second way to search through the elements of an array, using the `startsearch`, `anymore`, `nextelement`, and `donesearch` options. This approach is more general than the `foreach` approach just given, and in some cases it is more efficient, but it is more verbose than the `foreach` approach and isn't needed very often. See the reference documentation for details.

The `array exists` command can also be used to test whether a particular variable is an array, and the `array get` and `array set` commands can be used to convert between arrays and dictionaries:

```
  set a(head) hat
  set a(hand) glove
  set a(foot) shoe
  set apparel [array get a]
⇒ foot shoe head hat hand glove
  array exists a
⇒ 1
  array exists apparel
⇒ 0
  array set b $apparel
  lsort [array names b]
⇒ foot hand head
```

3.8 The `incr` and `append` Commands

`incr` and `append` provide simple ways to change the value of a variable. `incr` takes two arguments, which are the name of a variable and an integer. `incr` adds the integer to the variable's value, stores the result back into the variable, and returns the variable's new value as the result:

```
  set x 43
  incr x 12
⇒ 55
```

The number can have either a positive or a negative value. It can also be omitted, in which case it defaults to `1`:

```
  set x 43
  incr x
⇒ 44
```

Both the variable's original value and the increment must be integer strings, in either decimal, octal (indicated by a leading `0`), or hexadecimal (indicated by a leading `0x`).

Prior to Tcl 8.5, attempting to increment a variable that didn't exist raised an error. As of Tcl 8.5, the behavior of `incr` changed to create the variable and set its value to the increment:

```
   incr y
⇒ 1
   incr z 42
⇒ 42
```

The `append` command adds text to the end of a variable. It takes two or more arguments, the first of which is the name of the variable; the remaining arguments are new text strings to add to the variable. It appends the new text to the variable and returns the variable's new value. The following example uses `append` to compute a table of squares:

```
   set msg ""
   foreach i {1 2 3 4 5} {
       append msg "$i squared is [expr $i*$i]\n"
   }
   set msg
⇒ 1 squared is 1
   2 squared is 4
   3 squared is 9
   4 squared is 16
   5 squared is 25
```

Note

The `append` command does not automatically insert space characters between the text strings appended to the variable's value. If you want a space character, you must explicitly include it as part of the text strings you provide as arguments.

Neither `incr` nor `append` adds any new functionality to Tcl, since the effects of these commands can be achieved in other ways. However, they provide simple ways to do common operations. In addition, `append` is implemented in a fashion that avoids character copying. If you need to construct a very large string incrementally from pieces, it is much more efficient to use a command such as

```
append x $piece
```

than a command such as

```
set x "$x$piece"
```

3.9 Removing Variables: `unset` and `array unset`

The `unset` command destroys variables. It takes any number of arguments, each of which is a variable name, and removes all of the variables. Future attempts to read the variables result in errors just as if the variables had never been set in the first place. The arguments to `unset` may be either simple variables, elements of arrays, or whole arrays, as in the following example:

```
unset a earnings(January) b
```

In this case the variables `a` and `b` are removed entirely and the `January` element of the `earnings` array is removed. The `earnings` array continues to exist after the `unset` command. If `a` or `b` is an array, all of the elements of that array are removed along with the array itself.
You can also delete elements from an array using the `array unset` command, which accepts the name of an array variable and a pattern as arguments. Those element names that match the pattern using the matching rules of the `string match` command (as described in Section 5.10) are deleted from the array.

3.10 Predefined Variables

The Tcl library automatically creates and manages several global variables. The reference documentation for `tclvars` describes all of these variables, but we’ll examine the more frequently used ones in this section.
When you invoke a `tclsh` or `wish` script file, the name of the script file is stored in the variable `argv0`, the command-line arguments are stored as a list in the variable `argv`, and the number of command-line arguments is stored in the variable `argc`. Consider the following `tclsh` script:

```
#!/usr/bin/env tclsh
puts "The command name is \"$argv0\""
puts "There were $argc arguments: $argv"
```

If you place this script in a file named `printargs`, make the file executable, and then invoke it from your shell, it will print out information about its arguments:

```
  printargs red green blue
⇒ The command name is "printargs"
⇒ There were 3 arguments: red green blue
```

The variable `env` is predefined by Tcl. It is an array variable whose elements are all of the process's environment variables. For example, the following command prints out the user's home directory, as determined by the `HOME` environment variable:

```
puts "Your home directory is $env(HOME)"
```

The variable `tcl_platform` is an array variable that contains several elements describing the platform on which the application is running, such as the name of the operating system, its current release number, and the machine's instruction set:

```
  puts $tcl_platform(platform)
⇒ unix
  puts $tcl_platform(os)
⇒ Darwin
  puts $tcl_platform(machine)
⇒ i386
```

This array can be quite useful when you are writing scripts that must run without change on both Windows and Unix. Based on the values in the array, you could execute any platform-specific code required for that platform.

Note

For versions of Mac OS prior to Mac OS X, the value of `tcl_platform(platform)` was `macintosh`. Starting with Mac OS X, the value is `unix` and the value of `tcl_platform(os)` is `Darwin`, as shown above. For scripts that use Tk, you might have dependencies on the windowing system (for example, X11 versus Mac OS Aqua), in which case you can use the `tk windowingsystem` command described in Chapter 29 to query the windowing system of the computer running the script.

3.11 Preview of Other Variable Facilities

Tcl provides a number of other commands for manipulating variables. These commands will be introduced in full after you've learned more about the Tcl language, but this section contains a short preview of some of the facilities.

The `trace` command ensemble can be used to monitor a variable so that a Tcl script gets invoked whenever the variable is set, read, or unset. Variable tracing is sometimes useful during debugging, and it allows you to create read-only variables. You can also use traces for *propagation* so that, for example, a database or screen display gets updated whenever the value of a variable changes. Variable tracing is discussed in Section 15.6.

The `global` and `upvar` commands can be used by a procedure to access variables other than its own local variables. These commands are discussed in Chapter 9.

The `namespace` command ensemble creates and manages namespaces, which are named collections of commands and variables. Namespaces encapsulate the commands and variables to ensure that they don't interfere with the commands and variables in other namespaces. Namespaces are discussed in Chapter 10.

4. Expressions

Expressions combine values (or *operands*) with *operators* to produce new values. For example, the expression `4+2` contains two operands, `4` and `2`, and one operator, `+`; it evaluates to `6`. Many Tcl commands expect one or more of their arguments to be expressions. The simplest such command is `expr`, which just evaluates its arguments as an expression and returns the result as a string:

```
  expr (8+4) * 6.2
⇒ 74.4
```

Another example is `if`, which evaluates its first argument as an expression and uses the result to determine whether or not to evaluate its second argument as a Tcl script:

```
if {$x < 2} {set x 2}
```

This chapter uses the `expr` command for all of its examples, but the same syntax, substitution, and evaluation rules apply to all other uses of expressions as well.

4.1 Commands Presented in This Chapter

This chapter discusses the `expr` command:

- `expr` *`arg`* `?`*`arg arg`* `...?`

Concatenates all the *`arg`* values (with spaces between), evaluates the result as an expression, and returns the expression's value.

4.2 Numeric Operands

Expression operands are normally integers or real numbers. Integer values may be specified in decimal (the normal case), in binary (if the first two characters of the operand are `0b`), in octal (if the first two characters of the operand are `0o`), or in hexadecimal (if the first two characters of the operand are `0x`). For example, these are all ways of expressing the same integer value: 335 (decimal), 0o517 (octal), 0x14f (hexadecimal), 0b101001111 (binary). The octal and binary prefixes were added in Tcl 8.5.

Note

For compatibility with older Tcl releases, an octal integer value is also indicated when the first character of the operand is simply 0, whether or not the second character is also o. Therefore, 0517 is also equivalent to decimal 335. But 092 is not a valid integer: the leading 0 causes the number to be read in octal, but 9 is not a valid octal digit. The safest way to force numeric values with possible leading zeros to be treated as decimal integers is with the scan command, discussed in Section 5.9. The page http://wiki.tcl.tk/498 on the Tcler’s Wiki shows some strategies for handling this, including the following procedure:

```
proc forceInteger { x } {
    set count [scan $x {%d %c} n c]
    if { $count != 1 } {
        return -code error "not an integer: \"$x\""
    }
    return $n
}
```

You can specify real operands using most of the forms defined for ANSI C, including the following examples:

2.1
7.91e+16
6E4
3.

Note

These same forms are allowable not just in expressions but anywhere in Tcl that an integer or real value is required.

Expression operands can also be non-numeric strings. String operands are discussed in Section 4.6.

4.3 Operators and Precedence

Table 4.1 lists all of the operators supported in Tcl expressions; they are similar to the operators for expressions in ANSI C. Black horizontal lines separate groups of operators with the same precedence, and operators with higher precedence appear in the table above operators with lower precedence. For example, `4*2<7` evaluates to `0` because the `*` operator has higher precedence than `<`. Except in the simplest and most obvious cases you should use parentheses to indicate the way operators should be grouped; this helps prevent errors by you or by others who modify your programs.

Table 4.1 Summary of Operators Allowed in Tcl Expressions

Syntax	Result	Operand types
`-a`	Negative of *a*	int, real
`+a`	Unary plus of *a*	int, real
`!a`	Logical NOT: `1` if *a* is zero, `0` otherwise	int, real
`~a`	Bit-wise complement of *a*	int
`a**b`	Exponentiation: *a* raised to the *b* power	int, real
`a*b`	Multiply *a* and *b*	int, real
`a/b`	Divide *a* by *b*	int, real
`a%b`	Remainder after dividing *a* by *b*	int
`a+b`	Add *a* and *b*	int, real
`a-b`	Subtract *b* from *a*	int, real
`a<<b`	Left-shift *a* by *b* bits	int
`a>>b`	Arithmetic right-shift *a* by *b* bits	int
`a<b`	`1` if *a* is less than *b*, `0` otherwise	int, real, string
`a>b`	`1` if *a* is greater than *b*, `0` otherwise	int, real, string
`a<=b`	`1` if *a* is less than or equal to *b*, `0` otherwise	int, real, string
`a>=b`	`1` if *a* is greater than or equal to *b*, `0` otherwise	int, real, string
`a==b`	`1` if *a* is equal to *b*, `0` otherwise	int, real, string
`a!=b`	`1` if *a* is not equal to *b*, `0` otherwise	int, real, string
`a eq b`	`1` if *a* is equal to *b*, `0` otherwise	string
`a ne b`	`1` if *a* is not equal to *b*, `0` otherwise	string
`a in b`	List containment: `1` if *a* is an element appearing in list *b*, `0` otherwise	*a*: string; *b*: list
`a ni b`	Negated list containment: `1` if *a* is not an element appearing in list *b*, `0` otherwise	*a*: string; *b*: list
`a&b`	Bit-wise AND of *a* and *b*	int
`a^b`	Bit-wise exclusive OR of *a* and *b*	int
`a|b`	Bit-wise OR of *a* and *b*	int
`a&&b`	Logical AND: `1` if both *a* and *b* are nonzero, `0` otherwise	int, real
`a||b`	Logical OR: `1` if either *a* is nonzero or *b* is nonzero, `0` otherwise	int, real
`a?b:c`	Choice: if *a* is nonzero then *b*, else *c*	*a*: int, real

Operators with the same precedence group from left to right. For example, `10-4-3` is the same as `(10-4)-3`; both evaluate to `3`.

4.3.1 Arithmetic Operators

Tcl expressions support the arithmetic operators `+`, `-`, `*`, `/`, `%`, and `**`. The `-` operator may be used either as a binary operator for subtraction, as in `4-2`, or as a unary operator for negation, as in `-(6*$i)`. The `/` operator truncates its result to an integer value if both operands are integers. `%` is the modulus operator; its result is the remainder when its first operand is divided by the second. Both of the operands for `%` must be integers. `**` is the exponentiation operator, introduced in Tcl 8.5, which raises the first operand to the power of the second operand.

Note

The `/` and `%` operators have a more consistent behavior in Tcl than in ANSI C. In Tcl the remainder is always greater than or equal to zero and it has an absolute value less than the absolute value of the divisor. ANSI C guarantees only the second property. In both ANSI C and Tcl, the quotient will always have the property that `(x/y)*y + x%y` is `x` for all `x` and `y`.

4.3.2 Relational Operators

The operators `<` (less than), `<=` (less than or equal), `>=` (greater than or equal), `>` (greater than), `==` (equal), and `!=` (not equal) are used for comparing two values. Each operator produces a result of `1` (true) if its operands meet the condition and `0` (false) if they don't. These operators can be used for string as well as numeric comparisons; however, see Section 4.6 for more information on string comparison.

4.3.3 Logical Operators

The logical operators `&&`, `||`, and `!` are typically used for combining the results of relational operators, as in the expression

```
($x > 4) && ($x < 10)
```

Each operator produces a `0` or `1` result. `&&` (logical AND) produces a `1` result if both its operands are nonzero, `||` (logical OR) produces a `1` result if either of its operands is nonzero, and `!` (NOT) produces a `1` result if its single operand is zero.

In Tcl, as in ANSI C, a zero value is treated as false and anything other than zero is treated as true. Tcl also accepts string representations of Boolean values: `false`, `no`, or `off` for false, and `true`, `yes`, or `on` for true. These string representations are case-insensitive and may be unambiguously abbreviated.

Whenever Tcl generates a true/false value, though, it always uses `1` for true and `0` for false.
As in C, the operands of `&&` and `||` are evaluated sequentially: if the first operand of `&&` evaluates to `0`, or if the first operand of `||` evaluates to `1`, then the second operand is not evaluated.

4.3.4 Bit-wise Operators

Tcl provides six operators that manipulate the individual bits of integers: `&`, `|`, `^`, `<<`, `>>`, and `~`. These operators require their operands to be integers. The `&`, `|`, and `^` operators perform bit-wise AND, OR, and exclusive OR: each bit of the result is generated by applying the given operation to the corresponding bits of the left and right operands. Note that `&` and `|` do not always produce the same results as `&&` and `||`:

```
   expr 8&&2
⇒ 1
   expr 8&2
⇒ 0
```

The operators `<<` and `>>` use the right operand as a shift count and produce a result consisting of the left operand shifted left or right by that number of bits. During left shifts zeros are shifted into the low-order bits. Right-shifting is always “arithmetic right shift,” meaning that it shifts in zeros for positive numbers and ones for negative numbers. This behavior is different from right-shifting in ANSI C, which is machine-dependent.
The `~` operand (“ones complement”) takes only a single operand and produces a result whose bits are the opposite of those in the operand: zeros replace ones and vice versa.

4.3.5 Choice Operator

The ternary operator `?:` may be used to select one of two results:

```
expr {($a < $b) ? $a : $b}
```

This expression returns the smaller of `$a` and `$b`. The choice operator checks the value of its first operand for truth or falsehood. If it is true (nonzero), the argument following the `?` is the result; if the first operand is false (zero), the third operand is the result. Only one of the second and third arguments is evaluated.

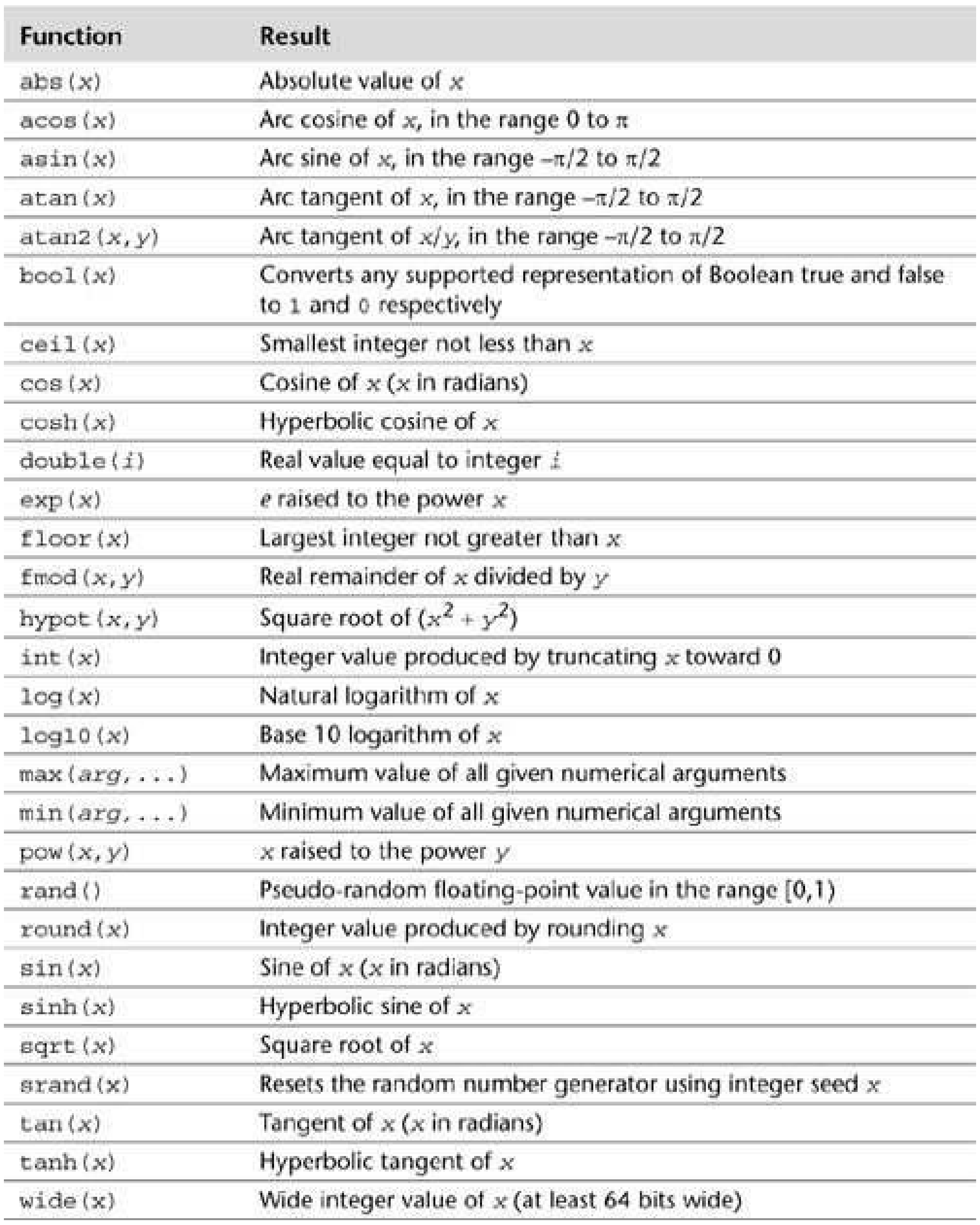
4.4 Math Functions

Tcl expressions support mathematical functions, such as `sin` and `exp`. Math functions are invoked using standard functional notation:

```
expr 2*sin($x)
expr hypot($x, $y) + $z
```

The arguments to math functions may be arbitrary expressions, and multiple arguments are separated by commas. See Table 4.2 for a list of all the built-in functions.

Table 4.2 Predefined Mathematical Functions Supported in Tcl Expressions



Function	Result
`abs(x)`	Absolute value of *x*
`acos(x)`	Arc cosine of *x*, in the range 0 to π
`asin(x)`	Arc sine of *x*, in the range –π/2 to π/2
`atan(x)`	Arc tangent of *x*, in the range –π/2 to π/2
`atan2(x,y)`	Arc tangent of *x*/*y*, in the range –π/2 to π/2
`bool(x)`	Converts any supported representation of Boolean true and false to `1` and `0` respectively
`ceil(x)`	Smallest integer not less than *x*
`cos(x)`	Cosine of *x* (*x* in radians)
`cosh(x)`	Hyperbolic cosine of *x*
`double(i)`	Real value equal to integer *i*
`exp(x)`	*e* raised to the power *x*
`floor(x)`	Largest integer not greater than *x*
`fmod(x,y)`	Real remainder of *x* divided by *y*
`hypot(x,y)`	Square root of $(x^2 + y^2)$
`int(x)`	Integer value produced by truncating *x* toward 0
`log(x)`	Natural logarithm of *x*
`log10(x)`	Base 10 logarithm of *x*
`max(arg,...)`	Maximum value of all given numerical arguments
`min(arg,...)`	Minimum value of all given numerical arguments
`pow(x,y)`	*x* raised to the power *y*
`rand()`	Pseudo-random floating-point value in the range [0,1)
`round(x)`	Integer value produced by rounding *x*
`sin(x)`	Sine of *x* (*x* in radians)
`sinh(x)`	Hyperbolic sine of *x*
`sqrt(x)`	Square root of *x*
`srand(x)`	Resets the random number generator using integer seed *x*
`tan(x)`	Tangent of *x* (*x* in radians)
`tanh(x)`	Hyperbolic tangent of *x*
`wide(x)`	Wide integer value of *x* (at least 64 bits wide)

As of Tcl 8.5, when the expression parser encounters a mathematical function such as `sin($x)`, it replaces the function internally with a call to an ordinary Tcl command in the `tcl::mathfunc` namespace. (Tcl's namespace feature is discussed in Chapter 10.) If the mathematical function contains comma-separated arguments, they are evaluated by `expr` and passed as separate arguments to the implementing procedure. Thus, the processing of an expression such as

```
expr {sin($x+$y)}
```

is the same in every way as the processing of

```
expr {[tcl::mathfunc::sin [expr {$x+$y}]]}
```

and

```
expr {atan2($y-0.3, $x/2)}
```

is equivalent to

```
expr {[tcl::mathfunc::atan2 [expr {$y-0.3}] [expr {$x/2}]]}
```

Note

The mechanism that maps functions to commands uses a relative namespace reference. If a namespace defines a command `[namespace current]::tcl:: mathfunc::sin`, calls to `sin` in expressions evaluated in that namespace resolve to it in preference to Tcl's predefined `::tcl::mathfunc::sin`.

This capability allows you to define your own mathematical functions simply by creating new commands in the `tcl::mathfunc` namespace. For example, to define a function to calculate the average of one or more numbers:

```
proc tcl::mathfunc::avg {args} {
  if {[llength $args] == 0} {
    return -code error \
      "too few arguments to math function \"avg\""
  }
  set total 0
  foreach val $args {
    set total [expr {$total + $val}]
  }
  return [expr {double($total) / [llength $args]}]
}
expr avg(2, 5, 1, 7)
⇒ 3.75
```

4.5 Substitutions

Substitutions can occur in two ways for expression operands. The first way is through the normal Tcl parser mechanisms, as in the following command:

```
expr 2*sin($x)
```

In this case the Tcl parser substitutes the value of the variable `x` before executing the command, so the first argument to `expr` will have a value such as `2*sin(0.8)`. The second way is through the expression evaluator, which performs an additional round of variable and command substitution on the expression while evaluating it. For example, consider the command

```
expr {2*sin($x)}
```

In this case the braces prevent the Tcl parser from substituting the value of `x`, so the argument to `expr` is `2*sin($x)`. When the expression evaluator encounters the dollar sign, it performs variable substitution itself, using the value of the variable `x` as the argument to `sin`.

Note

When the expression evaluator performs variable or command substitution, the value substituted must be an integer or real number (or a string, as described next). It cannot be an arbitrary expression.

Having two layers of substitution doesn't usually make any difference to the operation of the `expr` command, but it is vitally important for other commands such as `while` that evaluate an expression repeatedly and expect to get different results each time. For example, consider the following script, which finds the smallest power of 2 greater than a given number:

```
set pow 1
while {$pow<$num} {
    set pow [expr $pow*2]
}
```

The expression `$pow<$num` gets evaluated by `while` at the beginning of each iteration to decide whether or not to terminate the loop. It is essential that the expression evaluator use a new value of `pow` each time. If the braces were omitted so that variable substitution is performed while parsing the `while` command—for example, `while $pow<$num` . . .—then `while`'s argument would be a constant expression such as `1<44`; either the loop would never execute, or it would execute forever.

Note

Always quote your expressions with braces, even when using the `expr` command. Tcl is able to evaluate a braced expression much more efficiently than an unbraced expression. Quoting the expression with braces also prevents subtle security holes in your code. Consider a situation where a program prompts the user for a value, and then uses the value in an expression, such as

```
set x [expr $input + 2]
```

If a malicious user on a Windows system were to enter `[format C:\]`, the Tcl interpreter would substitute this string as the value of the `input` variable, and the expression evaluator would then execute the subcommand—formatting the C: drive. With the expression braced, the expression evaluator substitutes the value of the `input` variable; the result does not look like a valid arithmetic expression and so results in an error.

4.6 String Manipulation

Unlike expressions in ANSI C, Tcl expressions allow string operands for some operators, as in the following command:

```
if {$x eq "New York"} {
    ...
}
```

In this example, the expression evaluator compares the value of the variable `x` to the string `New York` using the string equality operator; the body of the `if` is executed if they are identical. Strings are compared lexicographically. The sorting order may vary from system to system.
To specify a string operand, you must either enclose it in quotes or braces or use variable or command substitution. It is important to enclose the entire expression in the preceding example in braces so that the expression evaluator substitutes the value of `x`. Consider the command

```
set result [expr $x eq "New York"]
```

If the braces are left out, the arguments to `expr` are concatenated, resulting in the expression

```
Los Angeles eq New York
```

The expression parser is not able to parse `Los` (it isn't a number, it doesn't make sense as a function name, and it can't be interpreted as a string because it doesn't have quotes around it), so a syntax error occurs.
If a string is enclosed in quotes, the expression evaluator performs command, variable, and backslash substitution on the characters between the quotes. If a string is enclosed in braces, no substitutions are performed. Braces nest for strings in expressions in the same way that they nest for words of a command.
In addition to `eq` and `ne`, which explicitly test for string equality and inequality, you can perform string comparisons with the `<`, `>`, `<=`, `>=`, `==`, and `!=` operators. However, these other operators do string comparisons only if one or both of the operands cannot be parsed as a number. For example, consider the following script:

```
    set x 8
    set y 010
    expr {$x == $y}
⇒ 1
```

Arithmetic comparison is used for the `expr` test, so the test evaluates to `1`.

Note

If you want to compare two values as strings in a situation where they may both look like numbers (e.g., the values are stored in variables so you don't know what their values are), you must use the string comparison operators or a command such as `string compare`, as described in Chapter 5.

4.7 List Manipulation

Tcl expressions also support two operators for list manipulation. (Chapter 6 discusses list manipulation in detail.) The `in` operator returns `1` if a particular string is an element in a list and `0` otherwise; the `ni` operator returns `1` if the string is not an element in the list and `0` otherwise. For example, the following tests whether `Los Angeles` appears as an element in the list stored in `cities`:

```
if {"Los Angeles" in $cities} {
    ...
}
```

The `in` and `ni` operators perform exact string comparison of the string against the list elements; they do no substring or pattern matching. Thus, the expression above is exactly equivalent to the following `lsearch -exact` test:

```
if {[lsearch -exact $cities "Los Angeles"] != -1} {
    ...
}
```

4.8 Types and Conversions

Tcl evaluates expressions numerically whenever possible. String operations are performed only for the string comparison operators and for the relational operators if one or both of the operands doesn't make sense as a number. Most operators permit either integer or real operands, but a few, such as `<<` and `&`, allow only integers.
Tcl supports arbitrarily large integers, but for performance reasons integers are usually stored internally with the C type `long`, which provides at least 32 bits of precision on most machines. Tcl automatically uses an arbitrary-precision integer data type internally for integer values unable to fit in a `long` value. Real numbers are represented with the C type `double`, which is usually represented with 64-bit values (about 15 decimal digits of precision) using the IEEE Standard for Binary Floating-Point Arithmetic.
If the operands for an operator have different types, Tcl automatically converts one of them to the type of the other. If one operand is an integer and the other is a real, the integer operand is converted to real. If one operand is a non-numeric string and the other is an integer or real, the integer or real operand is converted to a string.
Comparison operators always return a Boolean 0/1 result. The result of an arithmetic operation is a double if at least one of the operands is a double.

Integer arithmetic operations result in a native long integer if possible, or an arbitrary-precision integer otherwise. You can use the math function `double` to promote an integer to a real explicitly, and `int`, `wide`, `entier`, and `round` to convert a real value back to integer by truncation or rounding. The `int` function always returns a non-wide integer (converting by dropping the high bits). The `wide` function always returns a wide integer (converting by sign extending if the argument is a 32-bit number, or dropping the high bits if the argument is larger than a 64-bit number). The `entier` function coerces its argument to an integer of appropriate size. `entier` is distinguished from `int` and `wide` in that `int` results in an integer limited to the low-order bits of the machine word size and `wide` is limited to 64 bits, whereas `entier` results in whatever size of integer is needed to hold the full value.

4.9 Precision

Numbers are kept in internal form throughout the evaluation of an expression and are converted back to strings only when necessary. Integers are converted to signed decimal strings without any loss of precision. Real numbers are converted using as few digits as possible while still distinguishing any floating-point number from its nearest neighbors.

5. String Manipulation

This chapter describes Tcl's facilities for manipulating strings. Internally, Tcl stores strings as Unicode characters, but Tcl supports a variety of other character sets and can translate between character sets automatically and on demand. Tcl also supports handling binary strings. Tcl's string manipulation commands include pattern matching in two different forms: one that mimics the rules used by shells for file name expansion and another that uses regular expressions as patterns. Tcl also has commands for formatted input and output in a style similar to the C procedures `scanf` and `printf`. Finally, there are several utility commands for computing the length of a string, extracting characters from a string, case conversion, and other tasks.

5.1 Commands Presented in This Chapter

This chapter discusses the following commands for string manipulation:

- `binary format` *`format`* `?`*`value value`* `...?`

Returns a binary string whose layout is specified by *`format`* and whose contents come from the additional arguments.

- `binary scan` *`string format varName`* `?`*`varName varName`* `...?`

Parses fields from a binary *`string`*, returning the number of conversions performed. *`string`* is the input to be parsed, and *`format`* indicates how to parse it. Each *`varName`* gives the name of a variable; when a field is scanned from *`string`*, the result is assigned to the corresponding variable.

- `encoding convertfrom ?`*`encoding`*`?` *`string`*

Converts *`string`* to UTF-8 Unicode from the specified *`encoding`*. If you omit *`encoding`*, the system encoding is used.

- `encoding convertto ?`*`encoding`*`?` *`string`*

Converts *`string`* from UTF-8 Unicode to the specified *`encoding`*. If you omit *`encoding`*, the system encoding is used.

- `encoding names`

Returns a list of recognized encoding names.

- `encoding system ?`*`encoding`*`?`

Sets the system encoding to *`encoding`*. If you omit *`encoding`*, the command returns the current system encoding.

- `format` *`format`* `?`*`value value`* `...?`

Returns a result equal to `format` except that the `value` arguments have been substituted in place of `%` sequences in `format`.

- `regexp ?options? exp string ?matchVar? ?subVar subVar ...?`

Determines whether the regular expression `exp` matches part or all of `string` and returns `1` if it does, `0` if it doesn't. If there is a match, information about matching range(s) is placed in the variables named by `matchVar` and the `subVar`s, if they are specified. See the reference documentation for a complete list of `options`.

- `regsub ?options? exp string subSpec ?varName?`

Matches `exp` against `string` as for `regexp` and returns `1` if there is a match, `0` if there is none. Also copies `string` to the variable named by `varName`, making substitutions for the matching portion(s) as specified by `subSpec`. If you omit `varName`, the return value is the result of substitutions. See the reference documentation for a complete list of `options`.

- `scan string format varName ?varName varName ...?`

Parses fields from `string` as specified by `format`, places the values that match `%` sequences into variables named by the `varName` arguments, and returns the number of fields successfully parsed. Alternatively, if no `varName` arguments are provided, the values matched are returned as a list.

- `source ?-encoding encoding? fileName`

Reads and executes the script contained in the file `fileName`. Tcl reads the file using the system encoding unless you use `-encoding` to specify the `encoding`.

- `string bytelength string`

Returns the number of bytes used to represent the `string` internally. In most cases, you should use `string length` instead.

- `string compare ?-nocase? ?-length num? string1 string2`

Returns `-1`, `0`, or `1` if `string1` is lexicographically less than, equal to, or greater than `string2`. To ignore case, include the `-nocase` option. If you include `-length`, then only the first `num` characters are used for comparison.

- `string equal ?-nocase? ?-length num? string1 string2`

Returns `1` if `string1` and `string2` are identical, `0` otherwise. To ignore case, include the `-nocase` option. If you include `-length`, then only the first `num` characters are used for comparison.

- `string first string1 string2 ?startIndex?`

Returns the index in `string2` of the first character in the leftmost substring that exactly matches `string1`, or `-1` if there is no match. If `startIndex` is specified, then the search is constrained to start with the character in `string2` specified by the index.

- `string index string charIndex`

Returns the `charIndex`th character of `string`, or an empty string if there is no such character. The first character in `string` has index `0`.

- `string is class ?-strict? ?-failindex varname? string`

Returns `1` if `string` is a valid member of the specified character `class`; otherwise returns `0`. An empty `string` always returns `1` unless you include `-strict`, in which case it returns `0`. If you include `-failindex`, `varname` is assigned the index of the first character in `string` that was invalid for the specified `class`. The `varname` is not set if the command returns `1`. See the reference documentation for a list of recognized classes.

- `string last string1 string2 ?lastIndex?`

Returns the index in `string2` of the first character in the rightmost substring that exactly matches `string1`, or `-1` if there is no match. If `lastIndex` is specified, only the characters in `string2` at or before the specified `lastIndex` will be considered by the search.

- `string length string`

Returns the number of characters in `string`.

- `string map ?-nocase? mapping string`

Returns the new string created by replacing substrings in `string` based on the `mapping` dictionary. Each instance of a `mapping` key in the `string` is replaced with the corresponding `mapping` value. The `-nocase` option causes case-insensitive matching.

- `string match ?-nocase? pattern string`

Returns `1` if `pattern` matches `string` using glob-style matching rules (`*`, `?`, `[]`, and `\`) and `0` if it doesn't. The `-nocase` option causes case-insensitive matching.

- `string range string first last`

Returns the substring of `string` that lies between the indices given by `first` and `last`, inclusive.

- `string repeat string count`

Returns `string` repeated `count` number of times.

- `string replace string first last ?newstring?`

Returns the new string created by removing from `string` the range of characters from the `first` through `last` indices inclusively, replacing them with `newstring`, if specified.

- `string tolower string ?first? ?last?`

Returns a value identical to `string` except that all uppercase characters have been converted to lowercase. The entire `string` is converted unless you specify the index of the `first` and/or `last` characters to include in the conversion.

- `string totitle string ?first? ?last?`

Returns a value identical to `string` except that the first character is converted to title case and the rest are converted to lowercase. The entire `string` is converted unless you specify the index of the `first` and/or `last` characters to include in the conversion.

- `string toupper string ?first? ?last?`

Returns a value identical to `string` except that all lowercase characters have been converted to uppercase. The entire `string` is converted unless you specify the index of the `first` and/or `last` characters to include in the conversion.

- `string trim string ?chars?`

Returns a value identical to `string` except that any leading or trailing characters that appear in `chars` are removed. `chars` defaults to the whitespace characters.

- `string trimleft string ?chars?`

Same as `string trim` except that only leading characters are removed.

- `string trimright string ?chars?`

Same as `string trim` except that only trailing characters are removed.

- `string wordend string charIndex`

Returns the index of the character just after the last one in the word containing character `charIndex` of `string`. A word is either any contiguous range of letter, digit, and underscore characters, or any single character other than these.

- `string wordstart string charIndex`

Same as `string wordend` except returns the index of the first character in the word containing character `charIndex` of `string`.

5.2 Extracting Characters: `string index` and `string range`

Many string manipulation commands are implemented as options of the `string` command. For example, `string index` extracts a character from a string:

```
string index "Sample string" 3
⇒ p
```

The argument after `index` is a string, and the last argument gives the *index* of the desired character in the string. For all string commands, an index of `0` corresponds to the first character of the string, `1` corresponds to the second character, and so on. The index `end` refers to the last character of the string, `end-1` the next-to-last character, and so on. As of Tcl 8.5, you can also express an index by adding or subtracting two integer values. When using the `end±integer` or `integer±integer` format, you cannot include any whitespace characters in the index argument, even if you quote the argument. If the index is outside the range of the string, `string index` returns an empty string:

```
  set i 2
  string index "Sample string" end-$i
⇒ i
  string index "Sample string" 5+$i
⇒ s
  incr i
  string index "Sample string" 5+$i
⇒ t
```

The `string range` command is similar to `string index` except that it takes two indices and returns all the characters from the first index to the second, inclusive:

```
  string range "Sample string" 3 7
⇒ ple s
  string range "Sample string" 3 end
⇒ ple string
```

5.3 Length, Case Conversion, Trimming, and Repeating

The `string length` command counts the number of characters in a string and returns that number:

```
  string length "sample string"
⇒ 13
```

The `string toupper` command converts all lowercase characters in a string to uppercase, and the `string tolower` command converts all uppercase characters in its argument to lowercase:

```
  string toupper "Watch out!"
⇒ WATCH OUT!
  string tolower "15 Charing Cross Road"
⇒ 15 charing cross road
```

The `string` command provides three options for trimming: `trim`, `trimleft`, and `trimright`. Each option takes two additional arguments: a string to trim and an optional set of trim characters. The `string trim` command removes all instances of the trim characters from both the beginning and the end of its argument string, returning the trimmed string as result:

```
  string trim aaxxxbab abc
⇒ xxx
```

The `trimleft` and `trimright` options work in the same way, except that they remove the trim characters only from the beginning or the end of the string, respectively. The trim commands are most commonly used to remove excess whitespace; if no trim characters are specified, they default to the whitespace characters (space, tab, newline, carriage return, and form feed).
Another utility string command is `string repeat`, which returns a new string given an existing string and a repeat count:

```
  string repeat "*" 20
⇒ ********************
  string repeat "abc" 5
⇒ abcabcabcabcabc
```

5.4 Simple Searching

The command `string first` takes two additional string arguments as in the following example:

```
  string first th "There is the tub where I bathed today"
⇒ 9
  string first th "There is the tub where I bathed today"
⇒ 27
```

It searches the second string to see if there is a substring that is identical to the first string. If so, it returns the index of the first character in the leftmost matching substring; if not, it returns `-1`. The search starts from the beginning of the second string unless you also provide an argument specifying the character index where the search should start.
The command `string last` is similar except it returns the starting index of the rightmost matching substring:

```
  string last th "There is the tub where I bathed today"
⇒ 27
```

Note

You can perform more complex searches using regular expressions, as discussed in Section 5.11.

5.5 String Comparisons

The command `string compare` takes two additional string arguments and compares them, returning `0` if the strings are identical, `-1` if the first string sorts before the second, and `1` if the first string is after the second in sorting order:

```
  string compare Michigan Minnesota
⇒ -1
  string compare Michigan Michigan
⇒ 0
```

The `string equal` command does a simple string comparison of two strings, returning `1` if they are exactly the same and `0` if they aren't. Both `string compare` and `string equal` are case-sensitive unless you specify the `-nocase` option. You can also provide an optional `-length` option to specify that only the first *`length`* characters be used in the comparison:

```
  string equal cat cat
⇒ 1
  string equal dog Dog
⇒ 0
  string equal -nocase dog Dog
⇒ 1
  string equal -length 3 catalyst cataract
⇒ 1
```

5.6 String Replacements

You can perform a simple string replacement with the `string replace` command. It accepts a string as an argument along with the beginning and ending indices of a sequence of characters to delete, as well as an optional string argument to insert in their place:

```
  string replace "San Diego, California" 4 8 "Francisco"
⇒ San Francisco, California
  string replace "parsley, sage, rosemary, and thyme" 0 8
⇒ sage, rosemary, and thyme
```

The `string map` command maps values within a dictionary to string sequences within your text, replacing the sequences with the values from the dictionary. This can be useful as a general templating facility. The basic syntax is

string map *dictionary string*

The `string map` command replaces all instances of the *`dictionary`* keys in the *`string`* with their corresponding values to return a new string. Replacement is done in an ordered manner, so the key appearing first in the list is checked first, and so on. The string is iterated over only once, so earlier key replacements have no effect on later matches; for example:

```
  set entities {
    &   &amp;
    '   &apos;
    >   &gt;
    <   &lt;
   \"   &quot;
  }
  string map $entities {if (index > 0 && nbAtts == 0)}
⇒ if (index &gt; 0 &amp;&amp; nbAtts == 0)
```

With the `-nocase` option, the keys are replaced regardless of case; for example:

```
  string map -nocase \
    { RESOURCE "Ms. Ripley" CORPORATION "Weyland-Yutani" } \
   "Dear ResouRcE, welcome to your first day at corporation"
⇒ Dear Ms. Ripley, welcome to your first day at Weyland-Yutani
```

Note

You can perform more complex replacements using regular expressions, as discussed in Section 5.12.

5.7 Determining String Types

When manipulating strings, you often need to determine whether the value is in an appropriate format, such as whether it is numeric. To do so, use the `string is` command, which analyzes a string and returns `1` if the string is a member of a given class of characters and `0` otherwise; for example:

```
   string is digit 1234
⇒ 1
   string is digit "A man, a plan, a canal. Panama."
⇒ 0
```

By default, `string is` returns `1` for all character classes if the string is empty. Use the `-strict` option to force `string is` to return `0` if the string is empty:

```
   string is control ""
⇒ 1
   string is control -strict ""
⇒ 0
```

The `-failindex` option allows you to specify a variable that gets set if the test fails. In this case, the command sets the variable to the index in the string of the first character that fails the test; for example:

```
   string is digit -failindex idx "123c5"
⇒ 0
   puts $idx
⇒ 3
```

Table 5.1 lists the character classes supported by the `string is` command.

Table 5.1 Character Classes Supported by the `string is` Command

Class	Tests
`alnum`	Only Unicode alphabet or digit characters
`alpha`	Only Unicode alphabet characters
`ascii`	Only 7-bit ASCII characters
`boolean`	A recognized form of Boolean value (`0`, `false`, `no`, `off`, `1`, `true`, `yes`, or `on`)
`control`	Only Unicode control characters
`digit`	Only Unicode digits
`double`	A double-precision floating-point value (ignoring beginning or trailing spaces). If under- or overflow is detected, the `-failindex` variable is set to `-1`.
`false`	A recognized form of the Boolean false value (`0`, `false`, `no`, `off`)
`graph`	Only nonspace Unicode printing characters
`integer`	A 32-bit integer value (ignoring beginning or trailing spaces). If under- or overflow is detected, the variable with `-failindex` gets set to `-1`.
`list`	A valid list structure. If the list structure is incorrect, the `-failindex` variable is set to the first element in the list where the structure failed.
`lower`	Only lowercase Unicode characters
`print`	Only Unicode printing characters (includes space)
`punct`	Only Unicode punctuation characters
`space`	Only Unicode space characters
`true`	A recognized form of the Boolean true value (`1`, `true`, `yes`, or `on`)
`upper`	Only uppercase Unicode characters
`wideinteger`	A wide integer value (ignoring beginning or trailing spaces). If under- or overflow is detected, the `-failindex` variable is set to `-1`.
`wordchar`	Only alphanumeric or connecting punctuation characters (primarily the underscore)
`xdigit`	Only hexadecimal digit characters, including `0-9`, `a-f`, and `A-F`

Note

The `string is` command tests against the Unicode definition of characters. For example, Unicode digits include more than the ASCII characters 0 through 9.

5.8 Generating Strings with `format`

Tcl's `format` command provides facilities like those of the `sprintf` procedure from the ANSI C library. For example, consider the following command:

```
  format "The square root of 10 is %.3f" [expr sqrt(10)]
⇒ The square root of 10 is 3.162
```

The first argument to `format` is a format string, which may contain any number of conversion specifiers such as `%.3f`. For each conversion specifier, `format` generates a replacement string by reformatting the next argument according to the conversion specifier. The result of the `format` command consists of the format string with each conversion specifier replaced by the corresponding replacement string. In the preceding example, `%.3f` specifies that the next argument (the result of the `expr` command) is to be formatted as a real number with three digits after the decimal point. `format` supports almost all of the conversion specifiers defined for ANSI C `sprintf`, such as `%d` for a decimal integer, `%x` for a hexadecimal integer, and `%e` for real numbers in mantissa-exponent form. See the reference documentation for a complete list.
The `format` command plays a less significant role in Tcl than `printf` and `sprintf` play in C. Many of the uses of `printf` and `sprintf` are simply for conversion from binary to string format or for string substitution. Binary-to-string conversion isn't needed in Tcl because values are already stored as strings, and substitution is already available through the Tcl parser. For example, the command

```
set msg [format "%s is %d years old" $name $age]
```

can be written more simply as

```
set msg "$name is $age years old"
```

The `%d` conversion specifier in the `format` command could be written just as well as `%s`; with `%d`, `format` converts the value of `age` to a binary integer, then it converts the integer back to a string again.
`format` is typically used in Tcl to reformat a value to improve its appearance, or to convert from one representation to another (for example, from decimal to hexadecimal). As an example of reformatting, here is a script that prints the first ten powers of *e* in a table:

```
puts "Number  Exponential"
for {set i 1} {$i <= 10} {incr i} {
    puts [format "%4d %12.3f" $i [expr exp($i)]]
}
```

This script generates the following output on standard output:

```
Number  Exponential
   1        2.718
   2        7.389
   3       20.085
   4       54.598
   5      148.413
   6      403.429
   7     1096.630
   8     2980.960
   9     8103.080
  10    22026.500
```

The conversion specifier `%4d` causes the integers in the first column of the table to be printed right-justified in a field 4 digits wide, so that they line up under their column header. The conversion specifier `%12.3f` causes each of the real values to be printed right-justified in a field 12 digits wide, so that the values line up; it also sets the precision at 3 digits to the right of the decimal point.
The second main use for `format`, changing the representation of a value, is illustrated by the following script, which prints a table showing the ASCII characters that correspond to particular integer values:

```
puts "Integer  ASCII"
for {set i 95} {$i <= 101} {incr i} {
    puts [format "%4d       %c" $i $i]
}
```

This script generates the following output on standard output:

```
Integer  ASCII
  95       _
  96       `
  97       a
  98       b
  99       c
 100       d
 101       e
```

The value of `i` is used twice in the `format` command, once with `%4d` and once with `%c`. The `%c` specifier takes an integer argument and generates a replacement string consisting of the ASCII character represented by the integer.
An alternative to repeating the value of `i` in the preceding example is to use *positional specifiers* in the format string. If the `%` of a conversion specifier is followed by a decimal number and a `$`, as in `%2$d`, then the value to convert is

not taken from the next sequential argument. Instead, it is taken from the argument indicated by the number, where `1` corresponds to the first argument. With positional specifiers you can use the same argument as many times as desired. However, if there are any positional specifiers in the format string, all of the specifiers must be positional. So the following produces the same output as the previous script:

```
puts "Integer  ASCII"
for {set i 95} {$i <= 101} {incr i} {
    puts [format "%1$4d       %1$c" $i]
}
```

5.9 Parsing Strings with `scan`

The `scan` command provides almost exactly the same facilities as the `sscanf` procedure from the ANSI C library. `scan` is roughly the inverse of `format`. It starts with a formatted string, parses the string under the control of a format string, extracts fields corresponding to `%` conversion specifiers in the format string, and places the extracted values in Tcl variables. For example, after the following command is executed, variable `a` has the value `16` and variable `b` has the value `24.2`:

```
scan "16 units, 24.2% margin" "%d units, %f" a b
⇒ 2
```

The first argument to `scan` is the string to parse, the second is a format string that controls the parsing, and any additional arguments are names of variables to fill in with converted values. The return value of `2` indicates that two conversions were completed successfully.

`scan` operates by scanning the string and the format together. Each character in the format must match the corresponding character in the string, except for blanks and tabs, which are ignored, and `%` characters. A `%` encountered in the format indicates the start of a conversion specifier: `scan` converts the next input characters according to the conversion specifier and stores the result in the variable given by the next argument to `scan`. Whitespace in the string is skipped except in the case of a few conversion specifiers such as `%c`. See the reference documentation for a complete description of conversion specifier syntax.

One common use for `scan` is for simple string parsing, as in the preceding example. Another common use is for converting ASCII characters to their integer values, which is done with the `%c` specifier. The following procedure uses this feature to return the character that follows a given character in lexicographic ordering:

```
  proc next c {
       scan $c %c i
       format %c [expr {$i+1}]
  }
  next a
⇒ b
  next 9
⇒ :
```

The `scan` command converts the value of the `c` argument from an ASCII character to the integer used to represent that character, then the integer is incremented and converted back to an ASCII character with the `format` command.
Yet another typical use of `scan` is to force a string of digits with optional leading `0`s to be interpreted as decimal integers, as the leading `0`s usually cause Tcl to convert the string as an octal integer for arithmetic use. The following procedure forces a decimal interpretation of a numeric string:

```
proc forceDecimal {x} {
    set count [scan $x {%lld %c} n c]
    if {$count != 1} {
        error "not an integer: \"$x\""
     }
     return $n
  }
  set val 0987
  expr { $val + 1 }
∅ can't use invalid octal number as operand of "+"
  expr { [forceDecimal $val] + 1 }
⇒ 988
  forceDecimal xyz
∅ not an integer: "xyz"
```

The `format` argument of the `scan` command in the `forceDecimal` implementation is designed to detect any nondigit characters after the digits. If you were to use only `%d` as the format argument, the `scan` command would match the `123` in the string `123xyz`. The `ll` in the `%lld` conversion specifier is a size modifier supporting integers of unlimited precision. Without the `ll` modifier, the converted value would be constrained to the machine word size of the system on which the script executes:

```
  set val 012345678901234567890123456789 0
  expr { $val + 1 }
∅ can't use invalid octal number as operand of "+"
  expr { [forceDecimal $val] + 1 }
⇒ 123456789012345678901234567891
```

5.10 Glob-Style Pattern Matching

The simpler of Tcl's two forms of pattern matching is called "glob" style. It is named after the mechanism for file name expansion in Unix shells, which is called "globbing." Glob-style matching is easier to learn and use than the regular expressions described in the next two sections, but it works well only for simple cases. For more complex pattern matching you probably need to use regular expressions.
The command `string match` implements glob-style pattern matching:

string match ?-nocase? *pattern string*

The `string match` command returns `1` if the pattern matches the string, and `0` if it doesn't. For the pattern to match the string, each character of the pattern must be the same as the corresponding character of the string, except that a few pattern characters are interpreted specially. For example, a `*` in the pattern matches a substring of any length, so `Tcl*` matches any string whose first three characters are `Tcl`. Table 5.2 lists the special characters supported in glob-style matching. The match is case-sensitive unless you provide the `-nocase` option.

Table 5.2 Special Characters for Glob-Style Matching with the `string match` Command

Character(s)	Meaning
`*`	Matches any sequence of zero or more characters
`?`	Matches any single character
`[`*chars*`]`	Matches any single character in *chars*. If *chars* contains a sequence of the form *a*-*b*, any character between *a* and *b*, inclusive, will match.
`\`*x*	Matches the single character *x*. This provides a way to avoid special interpretation for any of the characters `* ? [ ] \` in the pattern.

The following examples illustrate the use of the `string match` command:

```
  string match a* alphabet
⇒ 1
  string match a* bat
⇒ 0
  string match {[ab]*} brown
⇒ 1
  string match a* Arizona
⇒ 0
  string match -nocase a* Arizona
⇒ 1
  string match {*\?} "Wow!"
⇒ 0
  string match {*\?} "What?"
⇒ 1
```

Many simple things can be done easily with glob-style patterns. For example, `*.[ch]` matches all strings that end with either `.c` or `.h`. However, many interesting forms of pattern matching cannot be expressed at all with glob-style patterns. For example, there is no way to use a glob-style pattern to test whether a string consists entirely of digits: the pattern `[0-9]` tests for a single digit, but there is no way to specify that there may be more than one digit.

5.11 Pattern Matching with Regular Expressions

Tcl's second form of pattern matching uses regular expressions, which are more complex than glob-style patterns but also more powerful. Tcl's regular expressions are based on Henry Spencer's publicly available implementation, and parts of the following description are copied from Spencer's documentation.

Tcl supports three flavors of regular expressions called *basic regular expressions* (BREs), *extended regular expressions* (EREs), and *advanced regular expressions* (AREs). BREs and EREs exist mostly for backward compatibility. The EREs are defined in POSIX, and the AREs are inspired at least a bit by Perl. Ever since Tcl 8.1, all Tcl commands support the ARE syntax by default; therefore, the rest of this book describes the use of the ARE syntax. See the reference documentation for more information on BREs and EREs.

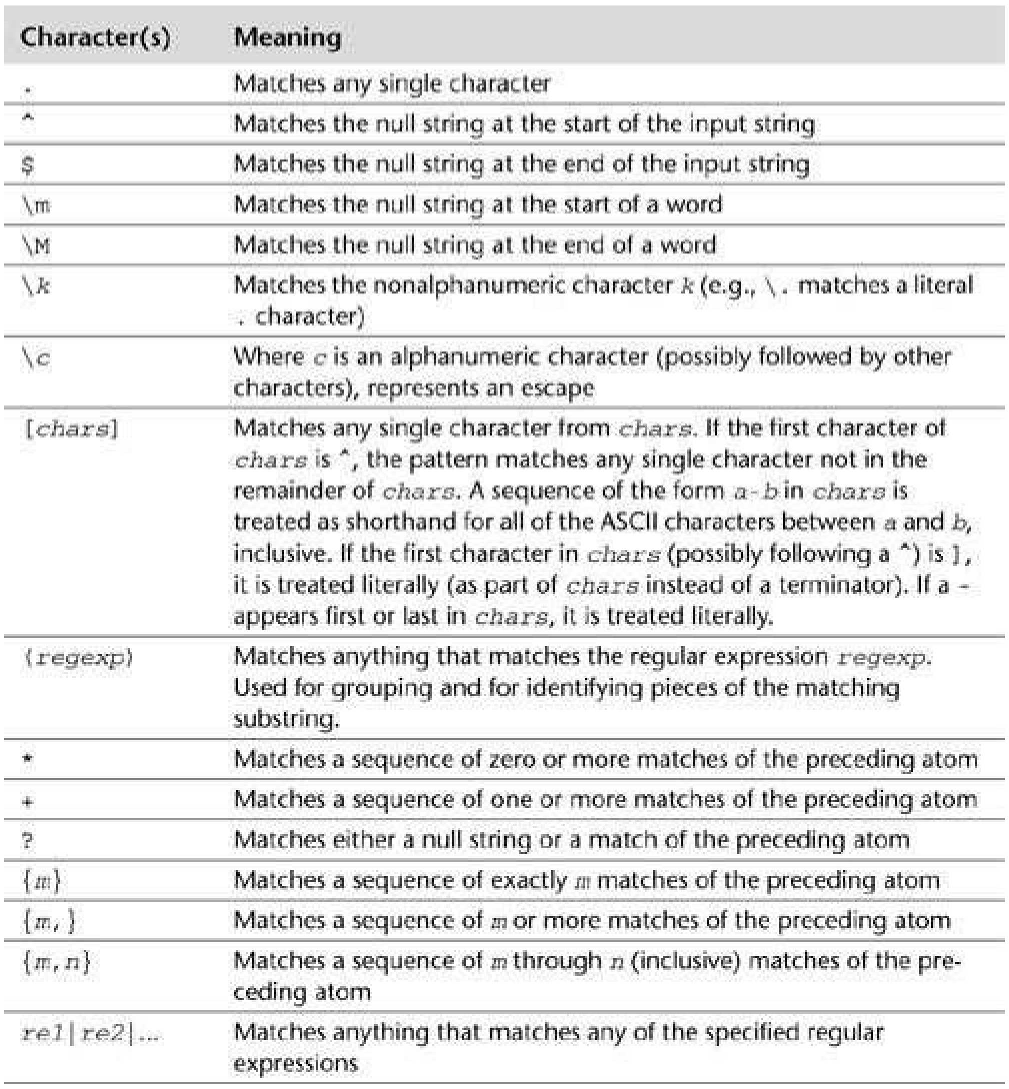
Note

Regular expression syntax can be quite complex. This book presents elementary regular expression syntax and use in Tcl. You should read the reference documentation for a complete description of Tcl's regular expression syntax. You can also learn more on web sites such as http://www.regular-expressions.info and http://regexlib.com. Additionally, the book *Mastering Regular Expressions, Third Edition* by Jeffrey E. F. Friedl (ISBN 0-596-52812-4) provides an indepth examination of regular expression syntax and use.

5.11.1 Regular Expression Atoms

A regular expression pattern can have several layers of structure. The basic building blocks are called *atoms*, and the simplest form of a regular expression consists of one or more atoms. For a regular expression to match an input string, there must be a substring of the input where each of the regular expression's atoms (or other components, as you'll see later) matches the corresponding part of the substring. In most cases atoms are single characters, each of which matches itself. Thus the regular expression `abc` matches any string containing `abc`, such as `abcdef` or `xabcy`.
A number of characters have special meanings in regular expressions; they are summarized in Table 5.3. The characters `^` and `$` are atoms known as *constraints* that match the position at the beginning and end of the input string, respectively. Thus `^abc` matches any string that starts with `abc`, `abc$` matches any string that ends in `abc`, and `^abc$` matches `abc` and nothing else. Similarly, the escape sequences `\m` and `\M` are constraints that match the beginning and end of a word, respectively. So `\mcat\M` matches only the word `cat` and not the `cat` in `catalog` or `concatenate`.

Table 5.3 Special Characters Permitted in Regular Expression Patterns



Character(s)	Meaning
`.`	Matches any single character
`^`	Matches the null string at the start of the input string
`$`	Matches the null string at the end of the input string
`\m`	Matches the null string at the start of a word
`\M`	Matches the null string at the end of a word
`\k`	Matches the nonalphanumeric character *k* (e.g., `\.` matches a literal `.` character)
`\c`	Where *c* is an alphanumeric character (possibly followed by other characters), represents an escape
`[chars]`	Matches any single character from *chars*. If the first character of *chars* is `^`, the pattern matches any single character not in the remainder of *chars*. A sequence of the form *a*-*b* in *chars* is treated as shorthand for all of the ASCII characters between *a* and *b*, inclusive. If the first character in *chars* (possibly following a `^`) is `]`, it is treated literally (as part of *chars* instead of a terminator). If a `-` appears first or last in *chars*, it is treated literally.
`(regexp)`	Matches anything that matches the regular expression *regexp*. Used for grouping and for identifying pieces of the matching substring.
`*`	Matches a sequence of zero or more matches of the preceding atom
`+`	Matches a sequence of one or more matches of the preceding atom
`?`	Matches either a null string or a match of the preceding atom
`{m}`	Matches a sequence of exactly *m* matches of the preceding atom
`{m,}`	Matches a sequence of *m* or more matches of the preceding atom
`{m,n}`	Matches a sequence of *m* through *n* (inclusive) matches of the preceding atom
`re1|re2|...`	Matches anything that matches any of the specified regular expressions

The atom `.` matches any single character, and the atom `\x`, where *x* is any single character, matches *x*. For example, the regular expression `.\$` matches any string that contains a dollar sign, as long as the dollar sign isn't the first character, and `\.$` matches any string that ends with a period.
Tcl's advanced regular expressions include an additional set of character-entry escapes, which also use the backslash character as a prefix. You can use these escape sequences as a means to specify otherwise hard-to-enter characters. Table 5.4 lists these special escapes.

Table 5.4 Regular Expression Character-Entry Escape Sequences

Escape	Represents
`\a`	The bell, or alert, character
`\b`	Backspace
`\B`	Backslash, same as \; used when needed to clean up regular expression syntax
`\cX`	Takes the lower 5 bits of the given character, *X*, and identifies the character with those lower 5 bits with all other bits zero
`\e`	Escape character
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Tab
`\uwxyz`	The Unicode character identified by the given four hexadecimal digits
`\v`	Vertical tab
`\xhhh`	The character whose ASCII value is represented by the given hexadecimal digits
`\0`	Null, or zero, character
`\xyz`	The character whose ASCII value is represented by the given two or three octal digits, as long as the value does not specify a back reference, as discussed in the reference documentation

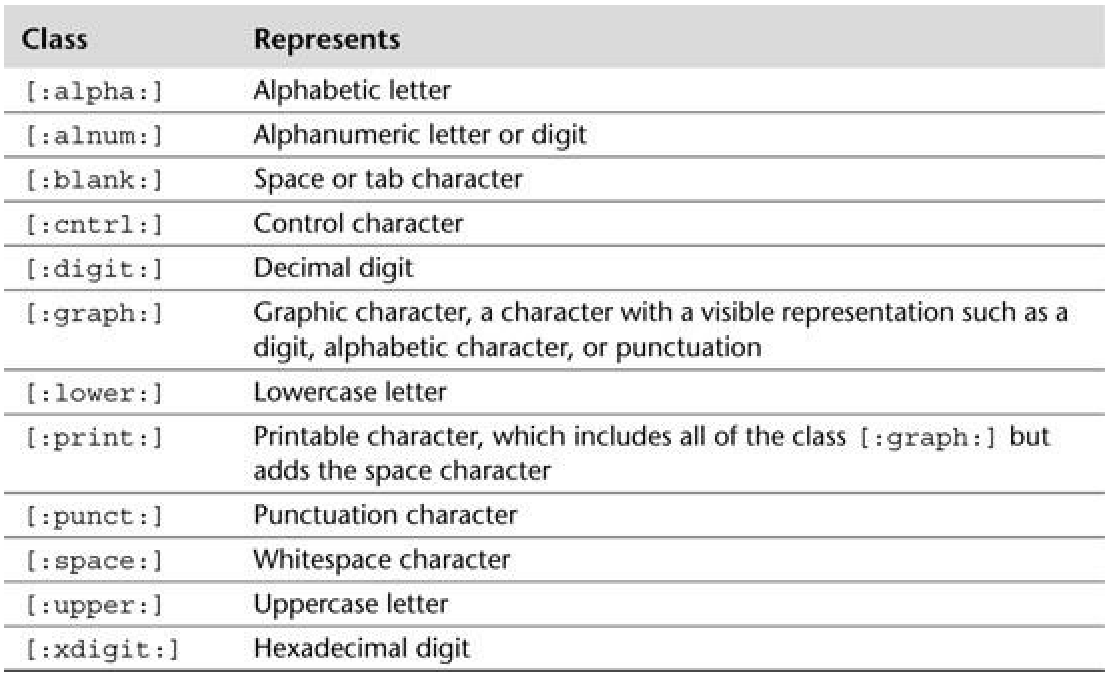
Besides the atoms already described, there are two other forms for atoms in regular expressions. The first form consists of any regular expression enclosed in parentheses, such as `(a.b)`. Parentheses are used for grouping, and a regular expression contained within a set of parentheses is often referred to as a *subexpression* or *subpattern*. They allow operators such as `*` to be applied to entire subexpressions as well as to atoms. They are also used to identify pieces of the matching substring for special processing. Both of these uses are described in more detail later.

The final form for an atom is a *range*, which is a collection of characters between square brackets. A range matches any single character that is one of the ones between the brackets. Furthermore, if there is a sequence of the form *a* - *b* among the characters, all of the ASCII characters between *a* and *b* are treated as acceptable. Thus, the regular expression `[0-9a-fA-F]` matches any string that contains a hexadecimal digit. If the character after the `[` is a `^`, the sense of the range is reversed: it matches only characters *not* among those specified between the `^` and the `]`.

Within the square brackets of a range, you also can use *character class* identifiers to represent all characters of a given class. For example, `[:alpha:]` within a range represents all alphabetic characters. Note that this goes beyond the idea of the ASCII letters `[A-Za-z]` and encompasses all alphabetic

Unicode characters. Table 5.5 lists the available classes of characters. Note that you can combine an explicit set of characters with one or more character classes within a range definition, as in

Table 5.5 Regular Expression Character Classes

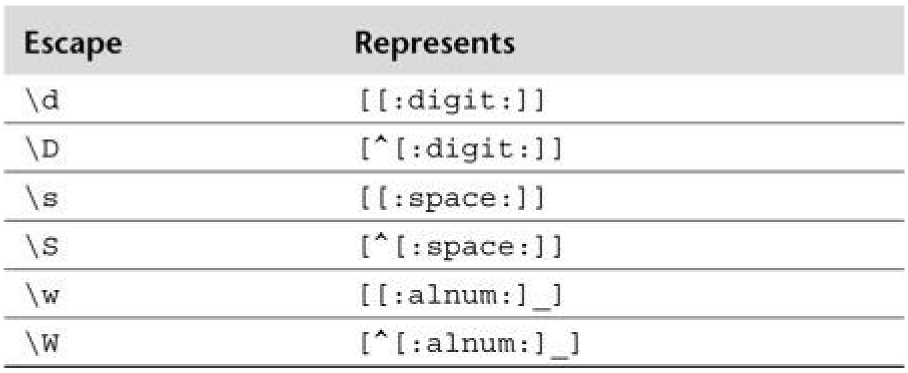


Class	Represents
`[:alpha:]`	Alphabetic letter
`[:alnum:]`	Alphanumeric letter or digit
`[:blank:]`	Space or tab character
`[:cntrl:]`	Control character
`[:digit:]`	Decimal digit
`[:graph:]`	Graphic character, a character with a visible representation such as a digit, alphabetic character, or punctuation
`[:lower:]`	Lowercase letter
`[:print:]`	Printable character, which includes all of the class `[:graph:]` but adds the space character
`[:punct:]`	Punctuation character
`[:space:]`	Whitespace character
`[:upper:]`	Uppercase letter
`[:xdigit:]`	Hexadecimal digit

```
[[:alpha:][:blank:],;.!?]
```

As a convenience, you can also use one of the special escape sequences listed in Table 5.6 as a shorthand for various character classes. Note that the `\w` "word character" and its inverse `\W` escape sequences include _ as well as alphanumeric characters in their definitions.

Table 5.6 Regular Expression Character Class Escape Sequences



Escape	Represents
`\d`	`[[:digit:]]`
`\D`	`[^[:digit:]]`
`\s`	`[[:space:]]`
`\S`	`[^[:space:]]`
`\w`	`[[:alnum:]_]`
`\W`	`[^[:alnum:]_]`

5.11.2 Regular Expression Branches and Quantifiers

Regular expressions may be joined with the `|` operator. The resulting regular expression matches anything that matches any of the regular expressions separated by the `|`, which are referred to as *branches*. For example, the following pattern matches a string containing `this`, `that`, or `other`:

```
this|that|other
```

Note that each branch may be any regular expression, including subexpressions, so it is possible to build quite complex structures. If two or more branches can match, Tcl prefers the longer match. To limit the extent of the branch definitions, you can enclose the set of branches within parentheses. For example, the following matches the string `this` or the string `that car`:

```
this|that car
```

In comparison, the following matches the string `this` or `that` followed by `car`:

```
(this|that) car
```

The operators `*`, `+`, and `?` are known as *quantifiers* and may follow an atom to specify repetition. An atom followed by `*` matches a sequence of zero or more matches of that atom. An atom followed by `+` matches a sequence of one or more matches of the atom. An atom followed by `?` matches either an empty string or a match of the atom. For example, `^(0x)?[0-9a-fA-F]+$` matches strings that are proper hexadecimal numbers, that is, those consisting of an optional `0x` followed by one or more hexadecimal digits.
Another set of quantifiers, known as *bounds*, allows you to specify exactly how many occurrences of an atom to match. Within a set of braces, you can specify the minimum and optionally the maximum number of occurrences, expressed as integers ranging from 0 to 255. `{`*`num`*`}` matches exactly *`num`* occurrences of an atom, `{`*`min`*`,}` matches at least *`min`* occurrences, and `{`*`min`*`,` *`max`*`}` matches at least *`min`* but no more than *`max`* occurrences.
When a regular expression can match more than one substring of a given string, the regular expression matches the one starting earliest in the string. Starting from that point, it then matches either the longest or the shortest substring possible, depending on the regular expression *preference*. By

default, the regular expression quantifiers are *greedy*, matching as many characters as possible. Any quantifier can be made *non-greedy*, and prefer the shortest substring possible, by following the quantifier with the `?` character. Thus, `+` and `{3,7}` are greedy quantifiers, whereas `+?` and `{3,7}?` are non-greedy quantifiers.

Note:

In Tcl, you cannot mix greedy and non-greedy behavior within a single regular expression. The preference for the entire regular expression is determined by the preference of the first quantifier within the expression. Processing ambiguities can arise if mixed greediness is allowed, and so Tcl's regular expression implementation was designed intentionally to disallow it.

5.11.3 Back References

In a regular expression, a *back reference* matches the same set of characters as matched by a previous subexpression. The syntax for a back reference is a `\` followed by the number of the preceding subexpression. Subexpressions are numbered in order of their opening (left) parentheses. For example, the expression `([ab])\1` matches the string `aa` or `bb`, but not `ab` or `ba`. To illustrate, consider writing a regular expression that would match a string quoted in either single or double quotes. One approach would be to use two branches:

```
'.*?'|".*?"
```

Alternatively, you can "capture" the first quoting character with a subexpression, then use a back reference to match the corresponding close quote:

```
(['"]).*?\1
```

Another example is that of finding accidentally repeated words, such as *the the*:

```
\m(\w+)\M\s+\m\1\M
```

5.11.4 Non-capturing Subexpressions

Often you need subexpressions purely for “structural” purposes, such as delimiting branches or applying a quantifier to a sequence, but don’t need to use the subexpression for a back reference or for parsing, as discussed in the following sections. In these situations, you can use what’s often called a *non-capturing subexpression*, which has the form `(?: expression)`. For example, the two following regular expressions match strings in the same way:

```
((ab){1,3}|(cd)+) xyz
(?:(?:ab){1,3}|(?:cd)+) xyz
```

The advantage of non-capturing subexpressions is that the regular expression evaluates faster. The disadvantages are that a back reference cannot refer to a non-capturing subexpression, and the substring matching the non-capturing subexpression cannot be extracted into a separate match variable, as described in the next section.

5.11.5 The `regexp` Command

The `regexp` command invokes regular expression matching. In its simplest form it takes two arguments: the regular expression pattern and an input string. It returns `0` or `1` to indicate whether or not the pattern matches the input string:

```
  regexp {^[0-9]+$} 510
⇒ 1
  regexp {^[0-9]+$} -510
⇒ 0
```

Note

The pattern is enclosed in braces so that the characters `$`, `[`, and `]` are passed through to the `regexp` command instead of triggering variable and command substitution. It is almost always a good idea to enclose regular expression patterns in braces.

If `regexp` is invoked with additional arguments after the input string, each additional argument is treated as the name of a variable. The first variable is filled in with the substring that matches the entire regular expression. The second variable is filled in with the portion of the substring that matches the first capturing subexpression within the pattern; the third variable is filled in with the match for the next capturing subexpression; and so on. If there are more variable names than capturing subexpressions, the extra variables are set to empty strings. For example, after executing the command

```
regexp {([0-9]+) *([a-z]+)} "Walk 10 km" a b c
```

variable `a` has the value `10 km`, `b` has the value `10`, and `c` has the value `km`. This ability to extract portions of the matching substrings allows `regexp` to be used for parsing.

It is also possible to specify additional options to `regexp` before the regular expression argument. You can use the `-start` option followed by a character index into the string to instruct `regexp` to start looking for a match from that location. The `-all` option tells `regexp` to match the pattern as many times as possible in the string and return the total number of matches. A `-nocase` option specifies that alphabetic atoms in the pattern should match either uppercase or lowercase letters in the string.

The `-indices` option specifies that the additional variables should not be filled in with the values of matching substrings. Instead, each should be filled in with a list giving the first and last indices of the substring’s range within the input string. After the command

```
regexp -indices {([0-9]+) *([a-z]+)} "Walk 10 km" \
        a b c
```

the variable `a` has the value `5 9`, `b` has the value `5 6`, and `c` has the value `8 9`.

The `-inline` option causes `regexp` to return as a list the data that it would otherwise place in match variables. For example, the following command returns a three-element list, where the first element contains the characters matching the entire expression and subsequent elements contain the characters matching each capturing subexpression:

```
    regexp {([0-9]+) *([a-z]+)} "Walk 10 km"
⇒ {10 km} 10 km
```

The `-line` option enables newline-sensitive matching. With this option, `[^` bracket expressions and `.` never match newlines, the `^` atom matches an empty string after any newline in addition to its normal function, and the `$` atom matches an empty string before any newline in addition to its normal function. As an example, the following command returns a count of all lines beginning with optional spaces or tabs followed by the string `ERROR`:

```
regexp -all -line -- {^[[:blank:]]*ERROR:} $text
```

Note

The `regexp` command also supports a `--` option to explicitly mark the end of options. It is a good practice to always use the `--` option; otherwise your pattern could mistakenly be interpreted as another option if its first character is a `-`.

5.12 Using Regular Expressions for Substitutions

Regular expressions can also be used to perform substitutions using the `regsub` command. Consider the following example:

```
  regsub there "They live there lives" their x
⇒ 1
```

The first argument to `regsub` is a regular expression pattern, and the second argument is an input string, just as for `regexp`. Also, like `regexp`, `regsub` returns `1` if the pattern matches the string, `0` if it doesn't. However, `regsub` does more than just check for a match: it creates a new string by substituting a replacement value for the matching substring. The replacement value is contained in the third argument to `regsub`, and the new string is stored in the variable named by the final argument to `regsub`. Thus, after the preceding command completes, `x` has the value `They live their lives`. If the pattern had not matched the string, `0` would have been returned and `x` would have the value `They live there lives` (the variable is set whether or not substitution took place).
You can also provide one or more options preceding the regular expression argument, and many of the same options supported by `regexp` are also available for use with `regsub`. Normally `regsub` makes only a single

substitution, for the first match found in the input string. However, if `-all` is specified, `regsub` continues searching for additional matches and makes substitutions for all of the matches found. It then returns the number of substitutions made. For example, after the command

```
regsub -all a ababa zz x
```

`x` has the value `zzbzzbzz`. If `-all` were omitted, `x` would be set to `zzbaba`.
The `-nocase` option requests a case-insensitive match of alphabetic atoms in the pattern. The `-start` option specifies a character index offset into the string where the pattern matching begins. The `-line` option enables newline-sensitive matching, as with `regexp`. And you can use the `--` option to explicitly mark the end of options to the command.
In the preceding examples, the replacement string is a simple literal value. However, if the replacement string contains a `&` or `\0`, the `&` or `\0` is replaced in the substitution with the substring that matches the regular expression. If a sequence of the form `\`*n* appears in the replacement string, where *n* is a decimal number, the substring that matches the *n*th capturing subexpression is substituted instead of the `\`*n*. For example, the command

```
regsub -all -- a|b axaab && x
```

doubles all of the `a`'s and `b`'s in the input string. In this case it sets `x` to `aaxaaaabb`. Alternatively, the command

```
regsub -all -- (a+)(ba*) aabaabxab {z\2} x
```

replaces sequences of `a`'s with a single `z` if they precede a `b` but don't also follow a `b`. In this case `x` is set to `zbaabxzb`. Backslashes may be used in the replacement string to allow `&`, `\0`, `\`*n*, or backslash characters to be substituted verbatim without any special interpretation.

Note

It's usually a good idea to enclose complex replacement strings in braces as in the preceding example; otherwise the Tcl parser will process backslash sequences, and the replacement string received by `regsub` may not contain backslashes that are needed.

5.13 Character Set Issues

A *character encoding* is simply a mapping of characters and symbols used in written language into a binary format used by computers. For example, in the standard ASCII encoding, the uppercase *A* character from the Latin character set is represented by the byte value 0x41 in hexadecimal. Other widely used character encodings include ISO 8859-1, used by many European languages, Shift-JIS and EUC-JP for Japanese characters, and Big5 for Chinese characters.
The Unicode Standard is a uniform encoding scheme for virtually all characters used in the world's major written languages. The text elements include letters such as *w* or *M*, characters such as those used in Japanese Hiragana to represent syllables, and ideographs such as those used in Chinese to represent full words or concepts. For more information on the Unicode Standard, visit the Unicode web site at http://www.unicode.org.
UTF-8 is a standard *transformation format* for Unicode characters. It is a method of transforming all Unicode characters into a variable-length encoding of bytes; a single Unicode character can be represented by 1 or more bytes. The advantage of the UTF-8 standard is that it and the Unicode Standard were designed so that Unicode characters corresponding to the standard 7-bit ASCII set (up to ASCII value 0x7F in hexadecimal) have the same byte values in both UTF-8 and ASCII encoding. In other words, an uppercase *A* character is represented by the singlebyte value 0x41 in both UTF-8 and ASCII encoding.
As of version 8.1, Tcl represents all strings internally as Unicode characters in UTF-8 format. Tcl also ships with built-in support for several dozen common character encoding standards and can convert strings from one encoding to another. The `encoding names` command returns a list of recognized encodings. The reference documentation describes how to add support for additional encodings.

5.13.1 Character Encodings and the Operating System

The *system encoding* is the character encoding used by the operating system for items such as file names and environment variables. Text files used by text editors and other applications are usually encoded in the system encoding as well, unless the application that produced them explicitly saves them in another format (for example, if you use a Shift-JIS text editor on an

ISO 8859-1 system).
Tcl's built-in commands automatically convert strings from UTF-8 format to the system encoding and vice versa whenever they communicate with the operating system. For example, Tcl automatically handles any encoding conversion needed if you execute commands such as

```
glob *
set fd [open "Español.txt" w]
exec myprog << "¡Bienvenido a Tcl!\n"
```

By default, the Tcl `source` command reads files using the system encoding. If you have a script file stored in a character encoding other than the system encoding, you can use the `source -encoding` option to provide the encoding name:

```
source -encoding shiftjis script.tcl
```

Tcl attempts to determine the system encoding during initialization, based on the platform and locale settings. Tcl usually can determine a reasonable default system encoding based on these settings, but if for some reason it cannot, it uses ISO 8859-1 as the default.

Note

You can override the default system encoding used by Tcl with the `encoding system` command, but you should avoid using this command if at all possible. If you set Tcl's default system encoding to anything other than the actual encoding used by your operating system, Tcl will likely find it impossible to communicate properly with your operating system.

5.13.2 Encodings and Channel Input/output

When reading and writing data on a channel, you need to ensure that Tcl uses the proper character encoding for that channel. The default encoding for all newly opened channels is the same as the system encoding. In most cases, you don't need to do anything special to read or write data because most text files are created in the system encoding. You need to take special steps only when accessing files in an encoding other than the system

encoding (for example, reading a file encoded in Shift-JIS format when your system encoding is ISO 8859-1).
The `fconfigure -encoding` option allows you to specify the encoding for a channel. Thus, to read from a file encoded in Shift-JIS format, you should execute the following commands:

```
set fd [open $file r]
fconfigure $fd -encoding shiftjis
```

Tcl then automatically converts any text read from the file into standard UTF-8 format. Similarly, if you are writing to a channel, you can use `fconfigure -encoding` to specify the target character encoding, and Tcl will automatically convert strings from UTF-8 to that encoding on output.

5.13.3 Converting Strings to Different Encodings

You can convert a string to a different encoding using the `encoding convertfrom` and `encoding convertto` commands. The `encoding convertfrom` command converts a string from a specified encoding into UTF-8 Unicode characters; the `encoding convertto` command converts a string from UTF-8 Unicode into a specified encoding. In either case, if you omit the encoding argument, the command uses the current system encoding.
As an example, the following command converts a string representing the Hiragana character HA from EUC-JP encoding into a UTF-8 string:

```
set ha [encoding convertfrom euc-jp "\xA4\xCF"]
```

5.14 Message Catalogs

In addition to the issue of handling different character set encodings, another challenge faced in the development of internationalized applications is that of presenting localized interfaces to users and other applications. To assist in creating localized applications, Tcl ships with a `msgcat` package, which provides a set of functions for managing multilingual user interfaces. It allows you to define strings in a *message catalog*, which is independent from your application or package and which you can edit or localize without modifying the application source code.
The basic principle of the `msgcat` package is that you create a set of *message*

files, one for each supported language, containing localized versions of all the strings your application or package can display. Then in your application or package, instead of using a string directly, you call the `::msgcat::mc` command to return a localized version of the string you want.

5.14.1 Using Message Catalogs

To use message catalogs from within your application or package, you must first load the `msgcat` package with the command

```
package require msgcat
```

By default, Tcl attempts to determine the locale according to the user's environment, as discussed in the `msgcat` reference documentation. If Tcl can't determine the locale based on the user's environment, the locale defaults to C. Optionally, you can call the `::msgcat::mclocale` command to explicitly set the locale:

::msgcat::mclocale ? *newLocale*?

If you omit the *`newLocale`* argument, `mclocale` returns the current locale. A locale string consists of a language code, an optional country code, then an optional modifier, all separated with _ characters. The country and language codes are specified in standards ISO-639 and ISO-3166. For example, the locale `en` specifies English, whereas `en_US` specifies U.S. English and `en_GB` specifies U.K. English.

Note

The `mclocale` command can be useful for implementing features such as a language menu, where users can select the language they want your application to display.

The next step is to call `::msgcat::mcload` to load the appropriate message files. The `mcload` command requires as an argument a directory containing your message files. The format of the messages files is described in the next section.
After loading the message files, wherever in your script you would typically specify a string to display, use the `::msgcat::mc` command instead. The `mc`

command takes as an argument a source string and returns the translation of that string in the current locale.
The following code fragment demonstrates how you could use the `msgcat` package in a script:

```
package require msgcat

# Use the default locale as determined by the system.
# You could explicitly set the locale with a command such as
# ::msgcat::mclocale "en_GB"

# Load the message files.  In this example, they are stored
# in a subdirectory named "msgs" which is in the same
# directory as this script.

::msgcat::mcload \
    [file join [file dirname [info script]] msgs]

# Display a welcome message

puts [::msgcat::mc "Welcome to Tcl!"]
```

In this example, instead of directly displaying the message `Welcome to Tcl!`, the application calls `mc` to retrieve a localized version of the string. The string returned by `mc` depends on the current locale. For example, in the `es` locale `mc` could return the Spanish-language greeting `¡Bienvenido a Tcl!`
The `mc` command performs a "best match" search to return a translation string. If it doesn't find an exact match in the locale specified in the user's environment or by the `mclocale` command, it checks successively more general locales for a match. For example, if the locale is `en_GB_Funky`, the locales `en_GB_Funky`, `en_GB`, `en`, and `""` (the empty string, corresponding to the `ROOT` locale) are searched in order until a matching translation string is found. If no match is found in any of those locales for the current namespace, `mc` checks the same set of locales for a match in the parent namespace, and so on until the global namespace is reached. This allows child namespaces to "inherit" messages from their parent namespace. (Tcl namespaces are discussed in Chapter 10.)
If a translation string doesn't exist for any of the locales in any of the namespaces, `mc` executes the procedure `::msgcat::mcunknown`. The default behavior of `mcunknown` is to return the original string (`Welcome to Tcl!` in this case), but you can redefine it to perform any action you want.

5.14.2 Creating Localized Message Files

To use the `msgcat` package, you need to prepare a set of message files for your package or application, all contained within the same directory. The name of each message file is a locale specifier followed by the extension `.msg` (for example, `es.msg` for a Spanish message file or `en_gb.msg` for a U.K. English message file). These file names must be all lowercase. The only exception is if you provide a message file for the root locale `""`, in which case it must be named `ROOT.msg`.

Note

Defining a `ROOT.msg` message file provides a fallback translation for all locales in the event one is not found in a more specific locale message file. This can be especially useful if you use symbolic source strings like `file_notfound` in your calls to `mc`.

Each message file contains a series of calls to `::msgcat::mcset` and/or `::msgcat::mcmset` to set the translation strings for that language. The format of the `mcset` command is

::msgcat::mcset *locale src-string* ?*translation-string*?

The `mcset` command defines a locale-specific translation for the given *`src-string`*. If no *`translation-string`* argument is present, then the value of *`src-string`* is also used as the locale-specific translation string.
So, if American English is the "source language" for your application, an `en_gb.msg` file might contain commands such as

```
::msgcat::mcset en_GB "Welcome to Tcl!"
::msgcat::mcset en_GB "Select a color:" "Select a colour:"
```

Note that no translation string is provided for the first line, so the resulting "translation" for the `en_GB` locale is the same as the American source string, `Welcome to Tcl!` If you omitted this entry in the message file, then calling `mc` with the source string `Welcome to Tcl!` in the `en_GB` locale would result in `mcunknown` being called if no other translation were available using the "best match" process described in Section 5.14.1. Although the default behavior of `mcunknown` would produce the desired results (returning `Welcome to Tcl!`), you could run into problems if you override the behavior of `mcunknown`. Therefore, it is always safest to include a mapping for every source string in your

application, even if a particular locale doesn't require a "translation" for that string.
An equivalent Spanish-language message file, `es.msg`, might contain

```
::msgcat::mcset es "Welcome to Tcl!" "¡Bienvenido a Tcl!"
::msgcat::mcset es "Select a color:" "Elige un color:"
```

An alternative to using multiple `mcset` commands is to use the `::msgcat::mcmset` command:

::msgcat::mcmset *locale translation-dict*

The *`translation-dict`* is a Tcl dictionary structure, which is a whitespace-separated sequence of keys and associated values. (See Chapter 7 for more information on Tcl dictionaries.) The keys in the dictionary are your source strings, and the values are the translation strings for the locale. Keys and values containing whitespace characters should be quoted. The advantage of `mcmset` is that it can be significantly faster than a sequence of `mcset` calls. So, the preceding Spanish-language message file could be equivalently specified as

```
::msgcat::mcmset es {
    "Welcome to Tcl!"     "¡Bienvenido a Tcl!"
    "Select a color:"     "Elige un color:"
}
```

5.14.3 Using Conversion Specifiers in Source and Translation Strings

Sometimes you would like to insert one or more arguments into a localized string. In support of this, `msgcat` allows you to include conversion specifiers in the source and translation strings that are interpreted exactly as with the `format` command described in Section 5.8. You can then pass values for these conversion specifiers as additional arguments to the `mc` command.
As an example, consider an entry like this in a `fr.msg` message file:

```
::msgcat::mcset fr \
    "Directory contains %d files" "Il y a %d fichiers"
```

You could then use the translation in your application like this:

puts [::msgcat::mc "Directory contains %d files" $num]

Note:

Positional specifiers can be especially useful with message files, as the order in which values are used might differ from language to language.

5.14.4 Using Message Catalogs with Namespaces

As discussed in Section 5.14.1, when you execute `mc`, it performs a "best match" search to return a translation string, for example, searching for a matching translation in the `en_GB` locale before searching the `en` locale. It's important to note that namespaces come into play during the lookup as well. If no matching translation is found in the current namespace, `mc` checks the same set of locales for a match in the parent namespace, and so on until the global namespace is reached. This allows child namespaces to "inherit" messages from their parent namespace. It also allows you to provide a set of translations specific to the library or module that you are developing if it is encapsulated in a namespace. (Tcl namespaces are discussed in Chapter 10.)
Translations in a message file that you register with `mcset` and `mcmset` are associated with the global namespace unless you explicitly execute those commands within a `namespace eval` script. For example, if you are developing a library that uses the `Mylib` namespace, you could register Spanish-language translations associated with that namespace in an `es.msg` file with code such as

```
namespace eval Mylib {
    ::msgcat::mcmset es {
        "Welcome to Tcl!"    "¡Bienvenido a Tcl!"
        "Select a color:"    "Elige un color:"
    }
}
```

A corresponding U.K. English message file, `en_gb.msg`, would contain code such as

```
namespace eval Mylib {
    ::msgcat::mcmset en_GB {
        "Welcome to Tcl!"    "Welcome to Tcl!"
        "Select a color:"    "Select a colour:"
    }
}
```

5.15 Binary Strings

Tcl was designed to handle primarily textual data. The original assumption was that manipulation of structured binary data could be achieved by writing custom C functions that could then be exposed as custom Tcl commands. However, working with binary data is common enough that the `binary` command was added to Tcl to manage binary data.

The `binary format` command creates a binary string in a Tcl variable. In most cases, you then write the binary string to a channel, either to a file or to a network socket. With the `binary format` command, you specify a formatting string followed by the data to format:

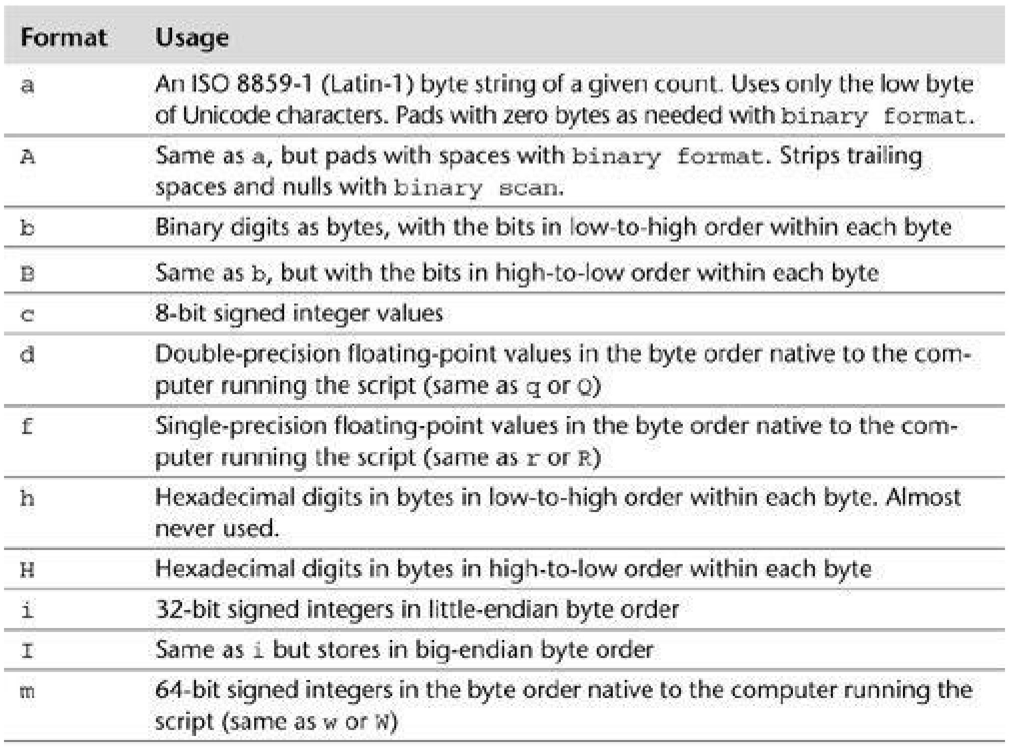
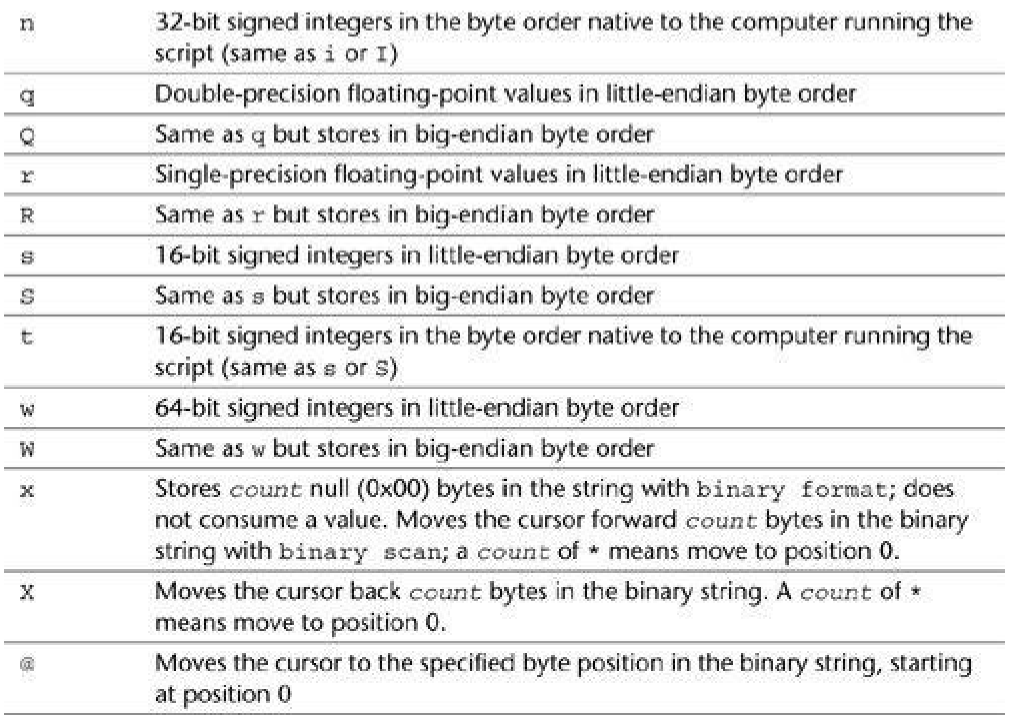
binary format *formatString ?arg arg ...?*

The *`formatString`* consists of a sequence of field specifiers, optionally separated by any number of spaces. Each field specifier consists of a character describing the type of data to format, optionally followed by a count of how many items of that type to format. The count defaults to `1` if omitted. For most types, a count of `*` indicates that all items in the associated argument should be used.

Table 5.7 lists the different format types supported by the `binary format` command. Most of the types pertain to storing numerical information; for example:

binary format c3 {1 2 120}

Table 5.7 Format Types for the `binary format` and `binary scan` Commands

Format	Usage
`a`	An ISO 8859-1 (Latin-1) byte string of a given count. Uses only the low byte of Unicode characters. Pads with zero bytes as needed with `binary format`.
`A`	Same as `a`, but pads with spaces with `binary format`. Strips trailing spaces and nulls with `binary scan`.
`b`	Binary digits as bytes, with the bits in low-to-high order within each byte
`B`	Same as `b`, but with the bits in high-to-low order within each byte
`c`	8-bit signed integer values
`d`	Double-precision floating-point values in the byte order native to the computer running the script (same as `q` or `Q`)
`f`	Single-precision floating-point values in the byte order native to the computer running the script (same as `r` or `R`)
`h`	Hexadecimal digits in bytes in low-to-high order within each byte. Almost never used.
`H`	Hexadecimal digits in bytes in high-to-low order within each byte
`i`	32-bit signed integers in little-endian byte order
`I`	Same as `i` but stores in big-endian byte order
`m`	64-bit signed integers in the byte order native to the computer running the script (same as `w` or `W`)
`n`	32-bit signed integers in the byte order native to the computer running the script (same as `i` or `I`)
`q`	Double-precision floating-point values in little-endian byte order
`Q`	Same as `q` but stores in big-endian byte order
`r`	Single-precision floating-point values in little-endian byte order
`R`	Same as `r` but stores in big-endian byte order
`s`	16-bit signed integers in little-endian byte order
`S`	Same as `s` but stores in big-endian byte order
`t`	16-bit signed integers in the byte order native to the computer running the script (same as `s` or `S`)
`w`	64-bit signed integers in little-endian byte order
`W`	Same as `w` but stores in big-endian byte order
`x`	Stores *`count`* null (0x00) bytes in the string with `binary format`; does not consume a value. Moves the cursor forward *`count`* bytes in the binary string with `binary scan`; a *`count`* of `*` means move to position 0.
`X`	Moves the cursor back *`count`* bytes in the binary string. A *`count`* of `*` means move to position 0.
`@`	Moves the cursor to the specified byte position in the binary string, starting at position 0

This command returns a binary string with the bytes `\x01\x02\x80`. The `c3` format string means to format three 8-bit signed integers. If you provide additional formats, each value passed is expected to provide the data for a given format; for example:

```
binary format I2a {1 -32 412534} E
```

The first format specifier, `I2`, takes only the first two numbers from the first argument, `{1 -32 41234}`, because of the count of `2`, and stores them as 32-bit signed integers in big-endian order (most significant byte first). Following that, the second format specifier, `a`, takes one character from the second value and stores it as an 8-bit Latin-1 character.
The `binary scan` command extracts data from a binary string:

> binary scan *binaryString formatString ?varName varName ...?*

The *`formatString`* consists of a sequence of field specifiers, optionally separated by any number of spaces. Each `binary scan` field specifier has the same format as with `binary format`, except that an optional `u` modifier can appear after the type character. The `u` indicates an unsigned value for integer fields; it's ignored for noninteger fields. For each field specifier that consumes values from the binary string, you must provide the name of a variable to store the results of the conversion; otherwise the command raises an error. If you provide more variable name arguments than required, the excess variables are left untouched by `binary scan`. If a field specifier includes a count, that number of values of the specified type are read from the binary string and stored as a list in the corresponding variable (except for character-oriented types). If there are not enough bytes of data in the binary string to fulfill all of the field specifiers, the corresponding variables are left untouched. The return value of `binary scan` is the number of variables set by the command.
Here are some examples of extracting data from a binary string:

```
  set data "\x40\x41\x42\x43\x44\x45\x46\x47\x48"
  binary scan $data a4c4 str chars
⇒ 2
  puts $str
⇒ @ABC
  puts $chars
⇒ 68 69 70 71
  binary scan $data a*c4 str2 chars2
⇒ 1
  puts $str2
⇒ @ABCDEFGH
  puts $chars2
∅ can't read "chars2": no such variable
  binary scan $data IS3 int shorts
⇒ 1
  puts $int
⇒ 1078018627
  puts $shorts
  can't read "shorts": no such variable
  binary scan $data IS2 int shorts
∅ 2
  puts $int
⇒ 1078018627
  puts $shorts
⇒ 17477 17991
  binary scan $data B8B5 bits1 bits2
⇒ 2
  puts $bits1
⇒ 01000000
  puts $bits2
⇒ 01000
  binary scan $data H* hex1
  puts $hex1
⇒ 404142434445464748
  binary scan $data h* hex2
  puts $hex2
⇒ 041424344454647484
```

6. Lists

Lists are used in Tcl to deal with collections of things, such as all the users in a group or all the files in a directory or all the options for a widget. Lists allow you to gather together any number of values in one place, pass around the collection as a single entity, and later get the component values back again. A list is an ordered collection of *elements* where each element can have any string value, such as a number, a person's name, the name of a window, or a word of a Tcl command. Lists are represented as strings with a particular structure; this means that you can store lists in variables, type them to commands, and nest them as elements of other lists.

6.1 Commands Presented in This Chapter

This chapter describes the structure of lists and presents more than a dozen basic commands for manipulating lists. The commands perform operations like creating lists, inserting and extracting elements, and searching for particular elements. Later chapters describe additional commands that take lists as arguments or return them as results.

- `concat ?`*`list list`*` ...?`

Joins multiple lists into a single list (each element of each *`list`* becomes an element of the result list) and returns the new list.

- `join `*`list`*` ?`*`joinString`*`?`

Concatenates list elements with *`joinString`* as the separator and returns the result. `joinString` defaults to a space.

- `lappend `*`varName value`*` ?`*`value`*` ...?`

Appends each *`value`* to the variable *`varName`* as a list element and returns the new value of the variable. Creates the variable if it doesn't already exist.

- `lassign `*`list varName`*` ?`*`varName`*` ...?`

Assigns successive elements from *`list`* to the variables given by the *`varName`* arguments in order. If there are more variable names than list elements, the remaining variables are set to the empty string. If there are more list elements than variables, a list of unassigned elements is returned.

- `lindex `*`list`*` ?`*`index`*` ...?`

Returns the *`index`*th element from *`list`* (`0` refers to the first element). With multiple index values, either as separate arguments or as a list, each index in turn selects an element from the previous indexing operation, allowing access to nested list elements.

- `linsert` *`list index value`* `?`*`value`* `...?`

Returns a new list formed by inserting all of the *`value`* arguments as list elements before the *`index`*th element of *`list`* (`0` refers to the first element).

- `list ?`*`value value`* `...?`

Returns a list whose elements are the *`value`* arguments.

- `llength` *`list`*

Returns the number of elements in *`list`*.

- `lrange` *`list first last`*

Returns a list consisting of elements *`first`* through *`last`* of *`list`*.

- `lrepeat` *`number value`* `?`*`value`* `...?`

Returns a list created by repeating the *`value`* arguments as elements for *`number`* occurrences.

- `lreplace` *`list first last`* `?`*`value value`* `...?`

Returns a new list formed by replacing elements *`first`* through *`last`* of *`list`* with zero or more new elements, each formed from one *`value`* argument.

- `lsearch ?`*`option`* `...?` *`list pattern`*

Searches for one or more elements in *`list`* that match *`pattern`*. The *`option`* arguments control the pattern-matching style (`-exact`, `-glob`, `-regexp`), whether to return element values (`-inline`) or indices, whether to match all (`-all`) or only the first occurrence in the list, and other behaviors. By default, performs glob matching, returning the index of the first match or `-1` if no element matches.

- `lset` *`varName`* `?`*`index`* `...?` *`newValue`*

Sets the element specified by *`index`* of the list stored in *`varName`* to *`newValue`*. Returns the new list stored in *`varName`*.

- `lsort ?`*`option`* `...?` *`list`*

Returns a new list formed by sorting the elements of *`list`*. The switches determine the comparison function and sorted order (default: `-ascii -increasing`).

- `split` *`string`* `?`*`splitChars`*`?`

Returns a list formed by splitting *`string`* at instances of *`splitChars`* and turning the characters between these instances into list elements.

6.2 Basic List Structure and the `lindex` and `llength` Commands

In its simplest form a list is a string containing any number of elements separated by any number of spaces, tabs, or newlines in any combination. For example, the string

```
John Anne Mary Jim
```

is a list with four elements. There can be any number of elements in a list, and each element can be an arbitrary string. In this simple form, elements cannot contain spaces, but there is additional list syntax that allows spaces within elements, as discussed later.
The `lindex` command extracts an element from a list:

```
lindex {John Anne Mary Jim} 1
⇒ Anne
```

`lindex` takes at least two arguments, a list and an index, and returns the selected element of the list. For all list commands, an index of `0` corresponds to the first element of the list, `1` corresponds to the second element, and so on; the index `end` refers to the last element of the list, `end-1` to the next-to-last element, and so on. As of Tcl 8.5, you can also express an index by adding or subtracting two integer values. When using the `end±`*`integer`* or *`integer±integer`* format, you cannot include any whitespace characters in the index argument, even if you quote the argument. If the index is outside the range of the list, `lindex` returns an empty string.
The `llength` command returns the number of elements in a list:

```
   llength {a b c d}
⇒ 4
   llength a
⇒ 1
   llength {}
⇒ 0
```

As you can see from the examples, a simple string like `a` is a proper list with one element, and an empty string is a proper list with zero elements.
When a literal list value is entered in a Tcl command, the list is usually enclosed in braces, as in the previous example. The braces are not part of the list; they are needed on the command line to pass the entire list to the command as a single word. When lists are stored in variables or printed out, there are no braces around them:

```
  set x {John Anne Mary Jim}
⇒ John Anne Mary Jim
```

Braces and backslashes within list elements are handled by the list commands in the same way that the Tcl command parser treats them in words. This means that you can enclose a list element in braces if it contains spaces, and you can use backslash substitution to get special characters such as braces into list elements; for example:

```
  set test1 {a b\ c d}
  llength $test1
⇒ 3
  lindex $test1 1
⇒ b c
  set test2 {a b\nc d}
  llength $test2
⇒ 3
  lindex $test2 1
⇒ b
  c
  set test3 {a \} b \{ c}
  llength $test3
⇒ 5
  lindex $test3 1
⇒ }
```

Note

When building a list whose elements contain unusual characters, the `list` command, discussed in the next section, is the safest way to ensure that the special characters receive proper quoting and escaping.

Braces are often used to nest lists within lists, as in the following example:

```
  lindex {a b {c d e} f} 2
⇒ c d e
```

In this case element `2` of the list is itself a list with three elements. There is no intrinsic limit on how deeply lists may be nested.
When manipulating nested lists, the `lindex` command allows you to specify one or more indices, either as separate arguments or as a list, to extract elements from sublists; for example:

```
  set elements {{a b} {c {d e f}} g}
  lindex $elements 1 1 2
⇒ f
  lindex $elements {0 0}
⇒ a
```

The first example is equivalent to

```
lindex [lindex [lindex $elements 1] 1] 2
⇒ f
```

though obviously shorter and less prone to errors.

6.3 Creating Lists: `list`, `concat`, and `lrepeat`

Tcl provides three commands that combine strings to produce lists: `list`, `concat`, and `lrepeat`. Each of these commands accepts an arbitrary number of arguments, and each produces a list as a result. However, they differ in the way they combine their arguments.
The `list` command joins its arguments so that each argument becomes a distinct element of the resulting list:

```
  list {a b c} {d e} f {g h i}
⇒ {a b c} {d e} f {g h i}
```

In this case, the result list contains only four elements. The `list` command always produces a list with proper structure, regardless of the structure of its arguments (it adds braces or backslashes as needed), and the `lindex` command can always be used to extract the original elements of a list created with `list`. The arguments to `list` need not themselves be well-formed lists.

Note

The `list` command is the safe way to create a list if you don’t know what the element values are (for example, if you’ve prompted the user for input or are reading a value from a file).

The `concat` command takes any number of lists as arguments and joins all of the elements of the argument lists into a single large list. If any of the elements in a list argument is a nested list, the element retains its nesting structure:

```
  concat {a b c} {d e} f {g h i}
⇒ a b c d e f g h i
  concat {a b} {c {d e f}}
⇒ a b c {d e f}
```

`concat` expects its arguments to have proper list structure; if the arguments are not well-formed lists, the result may not be a well-formed list either. In fact, all that `concat` does is to trim any leading and trailing whitespace characters from each of its argument strings and concatenate the result into one large string with space characters between the arguments. The same effect as `concat` can be achieved using double quotes:

```
  set x {a b c}
  set y {d e}
  set z [concat $x $y]
⇒ a b c d e
  set z "$x $y"
⇒ a b c d e
```

The `lrepeat` command creates a list by repeating a set of elements, each occurring as a separate argument, a specified number of times; for example:

```
  lrepeat 3 a
⇒ a a a
  lrepeat 4 a b c
⇒ a b c a b c a b c a b c
  lrepeat 3 {a b} c
⇒ {a b} c {a b} c {a b} c
```

6.4 Modifying Lists: `lrange`, `linsert`, `lreplace`, `lset`, and `lappend`

The `lrange` command returns a range of elements from a list. It takes as arguments a list and two indices, and it returns a new list consisting of the range of elements that lie between the two indices (inclusive):

```
  set x {a b {c d} e}
⇒ a b {c d} e
  lrange $x 1 3
⇒ b {c d} e
  lrange $x 0 1
⇒ a b
```

The `linsert` command forms a new list by adding one or more elements to an existing list:

```
  set x {a b {c d} e}
⇒ a b {c d} e
  linsert $x 2 X Y Z
⇒ a b X Y Z {c d} e
  linsert $x 0 {X Y} Z
⇒ {X Y} Z a b {c d} e
```

`linsert` takes three or more arguments. The first is a list, the second is the index of an element within that list, and the third and additional arguments are new elements to insert into the list. The return value from `linsert` is a list formed by inserting the new elements just before the element indicated by the index. If the index is `0`, the new elements go at the beginning of the list; if it is `1`, the new elements go after the first element in the old list; and so on. If the index is greater than or equal to the number of elements in the original list, the new elements are inserted at the end of the list.
The `lreplace` command deletes elements from a list and optionally adds new elements in their place. It takes three or more arguments. The first argument is a list and the second and third arguments give the indices of the first and last elements to be deleted. If only three arguments are specified, the result is a new list produced by deleting the given range of elements from the original list:

```
  lreplace {a b {c d} e} 3 3
⇒ a b {c d}
```

If additional arguments are specified to `lreplace`, as in the following example, they are inserted into the list in place of the elements that were deleted:

```
  lreplace {a b {c d} e} 1 2 {W X} Y Z
⇒ a {W X} Y Z e
```

Note

Tcl doesn't have an explicit command for deleting elements from a list, as the `lreplace` command provides this functionality if you don't provide any replacement element arguments.

A common operation is to update a list value stored in a variable by changing one of its elements. Historically, `lreplace` has been used for this purpose, as in

```
  set person {{Jane Doe} 30 female}
  set person [lreplace $person 1 1 31]
⇒ {Jane Doe} 31 female
```

Because `lreplace` does not modify the value of a variable directly, you must perform command substitution to execute it, and then assign the result as the new value of the variable. Not only is this verbose, it is also inefficient, as `lreplace` must copy the elements of the original list when creating the new list. This overhead can be pronounced when large lists are frequently updated.
The `lset` command is an efficient and concise method for changing the value of an element when the list is stored in a variable. It accepts the name of a variable, an index to an existing element—or a series of indices to an element in a nested sublist—and the new value to assign to the element. `lset` returns the variable's new value:

```
  lset person 1 32
⇒ {Jane Doe} 32 female
  lset person {0 1} Johnson
⇒ {Jane Johnson} 32 female
  lset person 0 0 Janice
⇒ {Janice Johnson} 32 female
```

Note

You cannot use the `lset` command to create new list elements. It can only modify existing elements. `lset` returns an error if the index refers to a nonexistent element.

The `lappend` command provides an efficient way to append new elements to a list stored in a variable. It takes as arguments the name of a variable and any number of additional arguments. Each of the additional arguments is appended to the variable's value as a new list element, and `lappend` returns the variable's new value:

```
  set x {a b {c d} e}
⇒ a b {c d} e
  lappend x XX {YY ZZ}
⇒ a b {c d} e XX {YY ZZ}
  puts $x
⇒ a b {c d} e XX {YY ZZ}
```

`lappend` is similar to `append` except that it enforces proper list structure. As with `append`, it isn't strictly necessary. For example, the command

```
lappend x $a $b $c
```

could be written instead as

```
set x [concat $x [list $a $b $c]]
```

However, as with `append`, `lappend` is implemented to optimize performance. For large lists, this can make a big difference.

Note

`lappend` and `lset` differ from other list commands such as `lreplace` in that the list is not included directly in the command; instead, you specify the name of a variable containing the list.

6.5 Extracting List Elements: `lassign`

The `lassign` command is a convenience for distributing the values of a list among one or more variables. The first argument is a list, and all subsequent arguments are the names of variables. `lassign` assigns successive elements from the list to the variables in order. If there are more variable names than list elements, the remaining variables are set to the empty string. If there are more list elements than variables, a list of unassigned elements is returned.

```
  lassign {a b c} x y z  ;# Produces an empty return value
  puts "<$x>, <$y>, <$z>"
⇒ <a>, <b>, <c>
  lassign {d e} x y z    ;# Produces an empty return value
  puts "<$x>, <$y>, <$z>"
⇒ <d>, <e>, <>
  lassign {f g h i} x y
⇒ h i
  puts "<$x>, <$y>"
⇒ <f>, <g>
```

Of course, similar results could be achieved with a series of `lindex` commands, though not as concisely:

```
set x [lindex $vals 0]
set y [lindex $vals 1]
set z [lindex $vals 2]
```

The behavior of `lassign` makes it easy to emulate the "shift" command of some languages:

```
set argv [lassign $argv nextArg]
```

Prior to Tcl 8.5, the `foreach` command was often used for its side effect of distributing list elements to individual variables. For example, to distribute the first three elements of the variable `coords` to three separate variables:

```
foreach {x y z} $coords { break }
```

The `break` command in this example serves as a "fail-safe" in case the list stored in `coords` consists of more than three elements.

6.6 Searching Lists: `lsearch`

The `lsearch` command searches a list for an element with a particular value. It takes two arguments, the first of which is a list and the second of which is a pattern:

```
  set x {John Anne Mary Jim}
  lsearch $x Mary
⇒ 2
  lsearch $x Phil
⇒ -1
```

`lsearch` returns the index of the first element in the list that matches the

pattern, or `-1` if there is no matching element.
One of three different pattern-matching techniques can be selected by specifying one of the switches `-exact`, `-glob`, or `-regexp` before the list argument:

```
   lsearch -glob $x A*
⇒ 1
```

The `-glob` switch causes matching to occur with the rules of the `string match` command described in Section 5.10. A `-regexp` switch causes matching to occur with regular expression rules as described in Section 5.11, and `-exact` insists on an exact match only. If no switch is specified, `-glob` is assumed by default. You can also negate the sense of the match with the `-not` option.
By default, `lsearch` finds only the first matching element. However, you can include the `-all` option to return all matching elements in the list:

```
   set states {California Hawaii Iowa Maine Vermont}
   lsearch -all $states *a
⇒ 0 2
```

The `-inline` option returns element values rather than indices. This is particularly useful when searching with patterns, as you then don't need to use `lindex` to extract the values in a separate step:

```
   lsearch -all -inline $states *ai*
⇒ Hawaii Maine
```

Note

Remember that if you want to detect the presence or absence of an exact string value as an element in a list, you can use the `in` and `ni` operators respectively in an expression. See Section 4.7 for more information.

6.7 Sorting Lists: `lsort`

The `lsort` command takes a list as an argument and returns a new list with the same elements, but sorted by default in increasing lexicographic order:

```
  lsort {John Anne Mary Jim}
⇒ Anne Jim John Mary
```

You can precede the list with any of several switches to control the sort. For example, `-decreasing` specifies that the result should have the "largest" element first; `-integer` and `-real` specify that the elements should be treated as integers or real numbers respectively and sorted according to value; `-dictionary` performs case-insensitive sorting and compares embedded digits as non-negative integers; and `-unique` discards all but the last occurrence of duplicated elements:

```
  lsort -decreasing {John Anne Mary Jim}
⇒ Mary John Jim Anne
  lsort {10 1 2}
⇒ 1 10 2
  lsort -integer {10 1 2}
⇒ 1 2 10
  lsort {Peach banana Apple pear}
⇒ Apple Peach banana pear
  lsort -dictionary {Peach banana Apple pear}
⇒ Apple banana Peach pear
  lsort -dictionary {n11.gif n1.gif n10.gif n9.gif}
⇒ n1.gif n9.gif n10.gif n11.gif
  lsort -unique {c a b q a z q}
⇒ a b c q z
```

If you have a nested list structure, the `-index` option allows you to specify the index of a sublist element on which to sort, rather than sorting on the entire subelement value:

```
  lsort -integer -index 1 {{First 24} {Second 18} {Third 30}}
⇒ {Second 18} {First 24} {Third 30}
```

Additionally, for lists containing data that can't be sorted lexicographically or numerically, you can use the `-command` option to specify your own sorting function (see the reference documentation for details).

6.8 Converting between Strings and Lists: `split` and `join`

The `split` command breaks a string into component pieces so that you can process the pieces independently. It creates a list whose elements are the pieces, so that you can use any of the list commands to process the pieces. For example, suppose a variable contains comma-separated values, and you want to convert it to a list with one element for each component:

```
   set x "Anita Sanchez,35,VP Marketing"
   set y 39,72,,-17,
   split $x ,
⇒ {Anita Sanchez} 35 {VP Marketing}
   split $y ,
⇒ 39 72 {} -17 {}
```

The first argument to `split` is the string to be split up, and the second argument contains one or more *split characters*. `split` locates all instances of any of the split characters in the string. It then creates a list whose elements consist of the substrings between the split characters. The ends of the string are also treated as split characters. If there are consecutive split characters, or if the string starts or ends with a split character as in the second example, empty elements are generated in the results. The split characters themselves are discarded.

Note

In practice, character-separated-value (CSV) data requires more careful handling, as the separator character might appear escaped as part of a value. Tcllib, the standard Tcl library, includes a `csv` package to handle CSV data properly. See Appendix B for more information on tcllib.

Several split characters can be specified, as in the following example:

```
   split xbaybz ab
⇒ x {} y z
```

If an empty string is specified for the split characters, each character of the string is made into a separate list element:

```
   split {a b c} {}
⇒ a { } b { } c
```

The `join` command is approximately the inverse of `split`. It concatenates list elements with a given separator string between them:

```
  join {{} usr include sys types.h} /
⇒ /usr/include/sys/types.h
  set x {24 112 5}
  expr [join $x +]
⇒ 141
```

`join` takes two arguments: a list and a separator string. It extracts all of the elements from the list and concatenates them with the separator string between each pair of elements. The separator string can contain any number of characters, including 0. In the first example here, a Unix-style path name is generated by joining the list elements with `/`. (The `file join` command described in Chapter 11 is a better way to create paths in this manner, as it is platform-independent.) In the second example, a Tcl expression is generated by joining the list elements with `+`.

6.9 Creating Commands as Lists

A very important relationship exists between lists and commands in Tcl. A well-formed Tcl command has the same structure as a list. A list evaluated as a Tcl script consists of a single command whose words are the list elements. In other words, the Tcl parser performs no substitutions whatsoever. It simply extracts the list elements, and each element becomes one word of the command. This property is very important because it allows you to generate Tcl commands that are guaranteed to parse in a particular fashion even if some of the command's words contain special characters such as spaces or `$`.
For example, suppose you are creating a button widget in Tk, and when the user clicks on the widget you would like to reset a variable to a particular value. You might create such a widget with the following command:

```
button .b -text Reset -command {set x 0}
```

The Tcl script `set x 0` is evaluated whenever the user clicks on the button. Now suppose that the value to be stored in the variable is not constant but instead is computed just before the `button` command and must be taken from a variable `initValue`. Furthermore, suppose that `initValue` could contain any string whatsoever. You might rewrite the command as

```
button .b -text Reset -command {set x $initValue}
```

The script `set x $initValue` is evaluated when the user clicks on the button.

However, this script uses the value of the global `initValue` variable at the time the user clicks on the button, which may not be the same as the value when the button was created. For example, the same variable might be used to create several buttons, each with a different intended reset value. Or, if the code to create the button were contained within a procedure, `initValue` might be a local variable that would not even exist at the time the user clicks the button, which would result in an error.
To solve this problem, you must generate a Tcl command that contains the *value* of the `initValue` variable, not its name, and use this as part of the `-command` option for the `button` command. Unfortunately, a simple approach like

```
button .b -text Reset -command "set x $initValue"
```

doesn't work in general. If the value of `initValue` is something simple like `47`, this works fine. The resulting command is `set x 47`, which produces the desired result. However, what if `initValue` contains `New York`? In this case the resulting command is `set x New York`, which has four words; `set` generates an error because there are too many arguments. Even worse, what if `initValue` contains special characters such as `$` or `[`? These characters could cause unwanted substitutions to occur when the command is evaluated.
The only solution that is guaranteed to work for any value of `initValue` is to use list commands to generate the command, as in the following example:

```
button .b -text Reset -command [list set x $initValue]
```

The result of the `list` command is a Tcl command whose first word is `set`, whose second word is `x`, and whose third word is the value of the `initValue` variable in scope at the time the button is created (not when it is pressed). For example, suppose that the value of `initValue` is `New York`. The command generated by `list` is

```
set x {New York}
```

which parses and executes correctly. Whatever value is present in `initValue` when the `button` command is invoked is assigned to `x` when the button is pressed, and this is guaranteed to work regardless of the contents of `initValue`. Any of the Tcl special characters are handled correctly by `list`:

```
  set initValue {Earnings: $1410.13}
  list set x $initValue
⇒ set x {Earnings: $1410.13}
  set initValue "{ \\"
  list set x $initValue
⇒ set x \{\ \\
```

7. Dictionaries

When you have a collection of things like a list but you wish to give each item a unique name and then access those items by their names, it is best to arrange the values into a dictionary. Like lists, dictionaries can contain any number of values, such as numbers, people's names, window names, channels, or command names. Dictionaries collect these values into a single entity that you can pass around like any other value, or nest inside other dictionaries or lists, and then you can retrieve those component values again by looking them up by their key name.

Dictionaries represent an ordered collection; dictionaries preserve the order of insertion of their keys and will iterate over them in that order. Dictionaries are represented as strings with a particular structure, looking just like a list with an even number of elements, and this means that you can store them in variables (including in array elements), type them into commands, and nest them inside themselves or lists. You can also put lists inside a dictionary, of course.

Dictionaries differ from arrays in several fundamental ways. Arrays are unordered collections of variables, not values, and may not be nested. This means that only arrays can have traces set on elements, and only dictionaries can be reliably iterated over in the same order or passed as values to other commands (in particular, non-global arrays require the use of `upvar` or explicit packing and unpacking).

Note

The dictionary data type was introduced in Tcl 8.5 as a formalization of existing good practice that was present in the strings managed by the `array` command's `get` and `set` subcommands, Tk widget options, and so on. The third-party `dict` extension provides a back port of most features of the `dict` command to Tcl 8.4, and earlier versions of Tcl can be supported through suitable scripts, though only 8.5 and later support efficient order-preserving dictionaries.

7.1 Commands Presented in This Chapter

This chapter describes the structure of dictionaries and presents the `dict` command, which manipulates dictionaries. The subcommands of `dict` can look up particular elements; insert, replace, and remove them; and list the names in the dictionary, among other things. Later chapters describe additional commands that use dictionaries as arguments or return them as results.

- `dict append` *`varName key value ?value ...?`*

Appends the given string values to the *`value`* associated with the *`key`* in the dictionary contained in *`varName`*.

- `dict create` *`key value ?key value ...?`*

Creates a dictionary from the given *`key`*s and *`value`*s. If a key occurs twice or more in the list of arguments, the value associated with the last instance of the key is used.

- `dict exists` *`dictionary key ?key ...?`*

Tests whether the *`key`* exists in the *`dictionary`*. Multiple keys may be given to test whether all the keys on a path through a group of nested dictionaries exist.

- `dict filter` *`dictionary filterType ...`*

Returns a new dictionary that is created from the supplied *`dictionary`* by applying the given filter. Filtering may be performed by matching (using `string match` rules) against keys or values or by using a script.

- `dict for {`*`keyVar valueVar`*`}` *`dictionary body`*

Iterates over the keys and values of the *`dictionary`*, setting the given variables to each of the keys and its associated value in turn, and then executing the *`body`* argument for each of them.

- `dict get` *`dictionary key ?key ...?`*

Returns the value in the dictionary with the given *`key`*. Multiple keys may be given to allow retrieval of a value from within nested dictionaries.

- `dict incr` *`varName key ?increment?`*

Increments the value in the dictionary in *`varName`* with the given *`key`*. If the increment is not given, it is `1`. If the key is not in the dictionary first, it is treated as if its value is `0`.

- `dict keys` *`dictionary ?pattern?`*

Returns a list of all the keys in the `dictionary`. If the *`pattern`* argument is specified, it returns only those keys that match the *`pattern`* according to the rules of `string match`.

- `dict lappend` *`varName key value ?value ...?`*

Appends the given list items to the value associated with the *`key`* in the dictionary contained in *`varName`*.

- `dict merge ?dictionary dictionary ...?`

Returns a new dictionary that is the combination of all the given dictionaries. Later key pairs override earlier key pairs when the keys correspond.

- `dict remove dictionary ?key ...?`

Returns a new dictionary that is the same as the supplied *dictionary*, except that each of the listed *key*s is not present in it. It is not an error if a key is given that is not present in the original *dictionary*.

- `dict replace dictionary ?key value ...?`

Returns a new dictionary that is the same as the supplied *dictionary*, except that the given set of *key* pairs will also form part of it (overriding any existing keys with the same names).

- `dict set varName key ?key ...? value`

Writes a new dictionary into the variable *varName* that is the same as the current contents of the variable, except that the given *key* will map to the given *value*. Multiple keys may be given to allow update of a value within a nested dictionary.

- `dict size dictionary`

Returns the size (number of keys) of the given dictionary.

- `dict unset varName key ?key ...?`

Removes the mapping for the given *key* from the dictionary in the variable *varName*. Multiple keys may be given to allow removal of a value within a nested dictionary. If the mapping is not present, no error will be generated.

- `dict update varName key localVar ?key localVar ...? body`

For each of the listed *key*s from the dictionary in *varName*, binds its value to *varName* while the *body* is being executed. When the *body* finishes executing, the variable contents are written back to the dictionary.

- `dict values dictionary ?pattern?`

Returns a list of values from the *dictionary*. If *pattern* is given, only those values that match it (according to the rules of `string match`) are returned.

- `dict with varName ?key ...? body`

Binds each of the *key*s in the dictionary in *varName* (or a subdictionary of it if a path of keys is given) to a local variable with the same name while *body* is being executed. When *body* finishes executing, the variable contents are written back to the dictionary.

7.2 Basic Dictionary Structure and the `dict get` Command

A dictionary is a structured value that looks just like a list with an even number of elements where the first, third, fifth (and so on) elements (the keys) are all distinct from each other. It is used to represent any collection of values that are indexed by strings, as opposed to lists, which are indexed by position. Dictionaries are used to represent structures (where the set of names is fixed) and maps (where the set of names is arbitrary). For example, the string

firstname Joe surname Schmoe title Mr

is a dictionary with three values (`Joe`, `Schmoe`, and `Mr`) named by the keys `firstname`, `surname`, and `title` respectively; it acts as a map from each of the keys to the value following it. There can be any number of elements in the dictionary, but each value must have a unique key. Both keys and values can be any arbitrary value.
The `dict get` command extracts an element from a dictionary:

```
    set example {firstname Joe surname Schmoe title Mr}
    dict get $example surname
⇒  Schmoe
```

`dict get` takes two arguments, a dictionary and a key to look up in the dictionary, and returns the value associated with that key; the command raises an error if the key does not have a value associated with it. The lookup of a value in a dictionary is very fast; behind the scenes, dictionaries are implemented as ordered hash tables, meaning that lookups are normally performed in nearly constant time.
Just as with lists, when a dictionary is entered in a Tcl script, it is usually enclosed in braces. For a complicated dictionary, it is often easier to use line breaks to separate key-value pairs:

```
    set prefers {
        Joe          {the easy life}
        Jeremy       {fast cars}
        {Uncle Sam}  {motherhood and apple pie}
    }
    dict get $prefers Joe
⇒  the easy life
```

In this case, the dictionary has *three* values (if it were a list, it would have six). You can also put dictionaries and lists inside each other as much as you wish; there is no limit imposed by the Tcl language. The next example maps from employee numbers to structures containing the details of the employees; both the map and the structures are dictionaries:

```
set employees {
    0001 {
        firstname Joe
        surname   Schmoe
        title     Mr
    }
    1234 {
        firstname Ann
        initial   E
        surname   Huan
        title     Miss
    }
}
puts [dict get [dict get $employees 1234] firstname]
```
⇒ *Ann*

This example could have been implemented using arrays with structured keys. But when arrays are used in that way, either the record relating to each employee would have been a list (requiring special caution when extracting or updating), or there would have been no record representing the employee data as a whole; whenever you would need such a record (for example, to pass to a procedure), you would have to extract it from the overall array, which would require much greater overhead. By contrast, nested dictionaries provide a much more flexible and efficient solution.
Indeed, there is a shortcut for getting a value out of a nested dictionary since the `dict get` command actually takes multiple keys as arguments, going along a path of nested dictionaries rather like a path name from a file system, like this:

```
puts [dict get $employees 1234 firstname]
```
⇒ *Ann*

The handling of nested dictionaries (including how to efficiently update them) is discussed in more depth in Section 7.6.
For comparison, here is an equivalent implementaton using arrays with structured keys, the most common method before Tcl 8.5:

```
  set employees(0001,firstname) Joe
  set employees(0001,surname)   Schmoe
  set employees(0001,title)     Mr
  set employees(1234,firstname) Ann
  set employees(1234,initial)   E
  set employees(1234,surname)   Huan
  set employees(1234,title)     Miss
  puts $employees(1234,firstname)
⇒ Ann
```

That seems simpler right up until you want to get Ann’s records together, which is much easier, clearer, and quicker with dictionaries:

```
  # The dictionary version
  set AnnsRecords [dict get $employees 1234]
⇒ firstname Ann initial E surname Huan title Miss
  # The array version
  set AnnsRecords [array get employees 1234,*]
⇒ 1234,title Miss 1234,initial E 1234,surname Huan 1234,firstname Ann
```

As the complexity of record processing increases, the filtering of the extra information from the element names can become seriously annoying. And because arrays are unordered, producing that output requires checking every element name in the whole array.

7.3 Creating and Updating Dictionaries

Tcl provides a command to help create dictionaries: `dict create`. This command is the analog of `list` (the constructor for lists); it takes an arbitrary number of key-value pairs and produces a dictionary with those key-value pairs in it. When a key occurs multiple times, the value last associated with the key is used. This is useful in situations when you want to create a dictionary in which either the keys or the values are not fixed at the time you execute the command.

```
  dict create  a b  c {d e}  {f g} h  a "b repeated"
⇒ a {b repeated} c {d e} {f g} hc {d e} a {b repeated} {f g} h
```

As you can see, the order of keys is the order in which they first appear in the list of arguments, but the value that is used is the last for a particular key in the sequence; earlier values for a duplicated key are dropped. This rule is applied when using `dict replace` to create a new dictionary based on the old one but with different values:

```
  set example [dict create firstname Ann initial E \
          surname Huan]
⇒ firstname Ann initial E surname Huan
  dict replace $example initial Y
⇒ firstname Ann initial Y surname Huan
  dict replace $example title Mrs surname Boddie
⇒ firstname Ann initial E surname Boddie title Mrs
```

As you can see, `dict replace` can also add to the collection of keys. If instead you want to produce a dictionary with some keys removed, use the `dict remove` command. Note that removing a key that did not previously exist is not an error.

```
  dict remove $example initial
⇒ firstname Ann surname Huan
  dict remove $example firstname title
⇒ initial E surname Huan
```

The `dict merge` command creates a new dictionary by merging two or more dictionaries, each provided as separate arguments. When two or more of the dictionaries have the same key, the resulting dictionary maps that key to the value according to the last dictionary on the command line containing a mapping for that key:

```
  set colors1 {foreground white background black}
  set colors2 {highlight red foreground green}
  dict merge $colors1 $colors2
⇒ foreground green background black highlight red
```

When you have a dictionary in a variable, you can update it directly to add keys, change what those keys map to, or remove keys. The first two operations are done with the `dict set` command, and the third operation is done with the `dict unset` command. The `dict set` command takes the name of the variable to update, the key to create or update, and the value that the key is to be set to and returns the resulting dictionary that it wrote back to the variable. The `dict unset` command takes the same arguments except for the value:

```
  set example [dict create firstname Ann initial E \
          surname Huan title Miss]
⇒ firstname Ann initial E surname Huan title Miss
  dict set example title Mrs
⇒ firstname Ann initial E surname Huan title Mrs
  dict get $example title
⇒ Mrs
  dict set example surname Boddie
⇒ firstname Ann initial E surname Boddie title Mrs
  dict get $example surname
⇒ Boddie
  dict unset example initial
⇒ firsname Ann surname Boddie title Mrs
  dict get $example initial
∅ key "initial" not known in dictionary
```

7.4 Examining Dictionaries: The `size`, `exists`, `keys`, and `for` Subcommands

Once you have a dictionary, there are a number of operations you may perform to examine it. One of the simplest is determining the number of elements in the dictionary. This is done using the `dict size` command:

```
  dict size {firstname Ann surname Huan title Miss}
⇒ 3
  dict size {}
⇒ 0
  set example {}
  dict set example a alpha
  dict set example b bravo
  dict set example c charlie
  dict set example d delta
  dict set example e epsilon
  dict size $example
⇒ 5
```

You can check whether a particular key is in a dictionary with the `dict exists` subcommand. This returns `1` when the `dict get` command can be used on the same dictionary and key to retrieve a value successfully, and `0` when `dict get` will fail because the dictionary doesn't contain the key:

```
  set example {title Miss firstname Ann surname Huan}
  dict exists $example firstname
⇒ 1
  dict exists $example initial
⇒ 0
```

To get a list of all the keys in the dictionary (in order), use the `dict keys` command. This command lists the keys of a dictionary value, optionally filtering them by a `string match`–style pattern. Continuing with the previous example:

```
  dict keys $example
⇒ title firstname surname
  dict keys $example {*name}
⇒ firstname surname
```

Similarly, the values from a dictionary can be retrieved in order using `dict values`, which also takes an optional pattern:

```
  dict values $example
⇒ Miss Ann Huan
  dict values $example *n*
⇒ Ann Huan
```

To loop over all the keys and values of a dictionary and execute some code for each of them, use the `dict for` command. This takes an argument listing a pair of variables (one for the key and one for the value associated with that key), a dictionary, and a Tcl script that forms the body of the loop. It returns the empty string. Just as with the `foreach` command, you can use `break` and `continue` to stop looping early or skip to the next key-value pair in the dictionary.
For example, to print out the contents of a dictionary neatly:

```
# Pretty print using the format command
dict for {key value} $dict {
    puts [format "%s: %s" $key $value]
}
# The result is empty, and the following lines are
# printed to the console
title: Miss
firstname: Ann
surname: Huan
```

It is possible to produce a sorted dictionary by taking advantage of the fact that dictionaries preserve the order of their keys. You do this by creating a second dictionary from the first that has the keys in sorted order, and then merging the values from the original dictionary with the `dict merge` subcommand:

```
    proc sortDict {dictionary} {
        set sorted {}
        foreach key [lsort [dict keys $dictionary]] {
            dict set sorted $key {}
        }
        return [dict merge $sorted $dictionary]
    }
    sortDict $example
  ⇒ firstname Ann surname Huan title Miss
```

7.5 Updating Dictionary Values

Sometimes it is necessary to update the values in a dictionary by changing them based on their current values rather than just replacing them with new ones. The `dict` command provides a number of convenience subcommands (based on other Tcl commands) to make this easier, so there is no need to get the value out of the dictionary into a variable, update the value in the variable, and then write that value back.
The easiest way to append a string or strings to a value in a dictionary is to use the `dict append` subcommand. This takes the name of a variable holding the dictionary you want to update, the key whose value you want to update, and one or more strings to append to the value and returns the updated dictionary as well as writing it back to the variable:

```
    set example {firstname Ann surname Huan title Miss}
    dict append example firstname ie
  ⇒ firstname Annie surname Huan title Miss
```

Similarly, when you want to build up a list in a dictionary value, you can use the `dict lappend` subcommand:

```
    set shopping {fruit apple veg carrot}
    dict lappend shopping fruit orange
  ⇒ fruit {apple orange} veg carrot
    dict lappend shopping fruit banana
  ⇒ fruit {apple orange banana} veg carrot
    dict lappend shopping veg cabbage beans
  ⇒ fruit {apple orange banana} veg {carrot cabbage beans}
```

Another updating operation supported by the `dict` command is `dict incr`. This takes a variable containing a dictionary, a key whose value will be incremented, and an optional value by which to increment the value. The result of the command is the updated dictionary, which is also written back to the variable. Analogously to the variable passed to the normal `incr` command (in Tcl 8.5 and later), the key does not have to exist previously in the dictionary, as it is assumed to have a value of `0` if absent. This is particularly useful when computing things like frequency histograms of words within a piece of text. An example is this procedure:

```
  proc computeHistogram {text} {
      set frequencies {}
      foreach word [split $text] {
          # Ignore empty words caused by double spaces
          if {$word eq ""} continue
          dict incr frequencies [string tolower $word]
      }
      return $frequencies
  }
  computeHistogram "this day is a happy happy day"
⇒ this 1 day 2 is 1 a 1 happy 2
```

Obviously, the `dict` command cannot supply a command for every kind of complex update that anyone might want to do. Instead, it supplies a general command for temporarily associating variables with selected keys of a dictionary, which you can use to build any kind of updating scheme at all. This is done using the `dict update` subcommand, which takes the name of a variable containing a dictionary, a list of keys in the dictionary and variables to associate with them, and a Tcl script that describes an operation that updates those variables. The result of the script is also the result of the overall `dict update` command. Upon completion of the script, the state of the variables is written back into the dictionary, thus allowing for arbitrarily complex updates of the dictionary.

Note

If a named key does not exist in the dictionary at the start of the `dict update` command, its corresponding variable is unset at the start of execution of the body script. At the end of the script, named keys that do not have their variables set are removed from the modified dictionary; that is, nonexistence of keys corresponds to nonexistence of variables.

Many different kinds of updates are possible using this mechanism. For example, here is how to reimplement the `dict unset` command:

```
dict update aDictionaryVariable $theKey localVar {
    unset localVar
}
```

More complex updates are also possible. Here is an example that switches the values between two keys:

```
    set example {firstname Ann surname Huan title Miss}
    dict update example firstname v1 surname v2 {
        lassign [list $v1 $v2] v2 v1
    }
    puts $example
⇒ firstname Huan surname Ann title Miss
```

This example shows how to make a square-a-value operation:

```
    proc squareValue {dictVar key} {
        upvar 1 $dictVar d
        dict update d $key v {
            set v [expr {$v ** 2}]
        }
    }
    set polyFactors {C 1 x 2 y 3}
    squareValue polyFactors y
⇒ C 1 x 2 y 9
```

One tricky feature is that updates to the variable containing the dictionary happen only when the body of the `dict update` command finishes, and the key-value pairs not mapped by this subcommand are left untouched. This makes the behavior of the system much easier to understand when a complex update is being performed:

```
  set example {firstname Ann surname Huan title Miss}
⇒ firstname Ann surname Huan title Miss
  set i "a dummy value"
  dict update example surname s notes n initial i {
      dict set example title Mrs
      unset s
      set n "have initial = [info exists i]"
      # Print the current contents of the dictionary
      puts $example
  }
⇒ firstname Ann surname Huan title Mrs
  # Get the contents of the variable after the dict
  # update command has completed; note it is different
  # in several respects, but the value for 'title' is
  # unchanged because that key was not listed at the
  # start of the command.
  puts $example
⇒ firstname Ann title Mrs notes {have initial = 0}
```

Note that although the `s` variable was removed during the body of the `dict update`, it was not until after the command had finished that the alterations were reincorporated back into the dictionary and the key `surname` was removed. Similarly, the key `initial` did not exist in the dictionary at the start and so the variable `i` was unset.

7.6 Working with Nested Dictionaries

Many of the `dict` subcommands have extra support for working with nested dictionaries. This allows you to specify multiple keys to the command and have the command go through the nested dictionaries to the point specified and work there, rather like a directory path name in a file system. The subcommands that support this method of working are `dict get`, `dict exists`, `dict set`, `dict unset`, and `dict with`.

The `dict get` command has the simplest nested dictionary behavior. When you ask for a value from a nested dictionary, it just goes through the nested dictionaries in order, using earlier keys to select the dictionaries where the later keys may be found. Thus, the command

```
dict get $dictionary keyOne keyTwo
```

is exactly the same as

```
dict get [dict get $dictionary keyOne] keyTwo
```

The `dict exists` command corresponds to the preceding line. It checks for the existence of all the keys along the path, and that all the steps along the path through the nested dictionaries are themselves dictionaries; it is an error if they exist but are not dictionaries. Thus, the following two commands are equivalent:

```
dict exists $dictionary keyOne keyTwo
expr {
    [dict exists $dictionary keyOne] &&
    [dict exists [dict get $dictionary keyOne] keyTwo]
}
```

As you can see, the multikey versions of both `dict get` and `dict exists` are much simpler to use when you have nested dictionaries.
The nested versions of the `dict set` and `dict unset` commands have even more complex equivalences to simple `dict` usage, enough that they are not described here in code. It is worth noting that the `dict set` command will create dictionaries along the path if necessary, though `dict unset` will only delete the key from the innermost dictionary on the path of keys; it does not delete the dictionaries forming the structure of the path, even if those dictionaries become empty.

```
  set nestedDict {firstname Ann surname Huan}
⇒ firstname Ann surname Huan
  dict set nestedDict address street {Ordinary Way}
⇒ firstname Ann surname Huan address {street {Ordinary Way}}
  dict set nestedDict address city Springfield
⇒ firstname Ann surname Huan address {street {Ordinary Way} city
  Springfield}
  dict get $nestedDict address street
⇒ Ordinary Way
  dict unset nestedDict address street
⇒ firstname Add surname Huan address {city Springfield}
```

The other command designed for working with nested dictionaries is the `dict with` subcommand. This allows the “opening out” of a dictionary into variables in a manner similar to the `dict update` command, but with a few differences. Instead of giving you control over which keys to work with and what variables they are to be bound to, `dict with` allows the selected dictionary or any subdictionary (as specified by the path of keys) to open out in its entirety.

```
set example {
    A {
        alphabet  {a alpha b bravo c charlie}
        animals   {cow calf sheep lamb pig ? goose ?}
    }
    C {
        comedians {laurel&hardy morecambe&wise}
    }
}
dict with example C {
    puts "comedians: $comedians"
    lappend comedians "steve martin"
}
⇒ comedians: laurel&hardy morecambe&wise
⇒ laurel&hardy morecambe&wise {steve martin}
  dict with example A alphabet {
      puts "NATO ABC: $a $b $c"
  }
⇒ NATO ABC: alpha bravo charlie
  dict with example A animals {
      set pig piglet
      set goose gosling
  }
  dict with example A {
      dict for {k v} $animals {
          puts "$k has baby $v"
      }
  }
⇒ cow has baby calf
⇒ sheep has baby lamb
⇒ pig has baby piglet
⇒ goose has baby gosling
```

The `dict with` command can be used to allow the stowing of some persistent procedure state in a global variable without polluting the global namespace with lots of different variables or requiring every use of the global state to be encapsulated within array syntax, which is the other solution. The use of this is illustrated within the `stepCounter` procedure in this example:

```
set counters {
    next 0
}
proc makeCounter {{step 1} {offset 0}} {
    global counters
    set id counter[dict get $counters next]
    dict incr counters next
    dict set counters $id [dict create \
            state 0 step $step offset $offset]
    return $id
}
proc stepCounter {id} {
    global counters
    dict with counters $id {
        return [expr {[incr state $step] + $offset}]
    }
}
```

When used, this makes for a very simple stateful counter system that can count in steps of any size desired with an arbitrary offset applied at each stage, and all at a cost of exactly one global variable. A sample of this simple-to-extend mechanism is shown below, where two counters are first created and then used in an interleaved fashion:

```
  set a [makeCounter]
⇒ counter0
  set b [makeCounter 5 -4]
⇒ counter1
  stepCounter $a
⇒ 1
  stepCounter $a
⇒ 2
  stepCounter $b
⇒ 1
  stepCounter $a
⇒ 3
  stepCounter $b
⇒ 6
  stepCounter $a
⇒ 4
  stepCounter $b
⇒ 11
  puts $counters
⇒ next 2 counter0 {step 1 state 4 offset 0} counter1 {step 5 state
  15 offset -4}
```

Another key usage of dictionaries and the `dict with` subcommand is for representing the results of a database query. Each column from the row of results can be given a unique name (e.g., the name of the table column) with the contents of that column being the dictionary value[1], and the iteration order of the dictionary being the order of the columns. For well-behaved query results (i.e., almost all) the dictionary is then trivial to map into variables using `dict with`.

1. Null columns are best represented as absent keys, since the `NULL` value actually represents an absence of a value for a particular column in that row. Of course, a database interface might also provide other options for handling Nulls in particular columns (e.g., particular strings). This is all outside the scope of dictionaries, though. They provide mechanisms; they do not define interface policy.

```
    set result [dbConn query {
        SELECT firstname, surname, title FROM staff
            LIMIT 2
    }]
⇒ {firstname Joe surname Schmoe title Mr} {firstname Annie surname
    Huan title Miss}
    foreach row $result {
        dict with row {
            puts "found $title $firstname $surname"
        }
    }
⇒ found Mr Joe Schmoe
⇒ found Miss Annie Huan
```

8. Control Flow

This chapter describes the Tcl commands for controlling the flow of execution in a script. Tcl's control flow commands are similar to the control flow statements in the C programming language and the Unix shell `csh`, including `if`, `while`, `for`, `foreach`, `switch`, and `eval`.

8.1 Commands Presented in This Chapter

You can use the following commands to control the flow of execution in a Tcl script:

- `break`

Terminates the innermost nested looping command.

- `continue`

Terminates the current iteration of the innermost looping command and goes on to the next iteration of that command.

- `eval` *`arg`* `?`*`arg arg`* `...?`

Concatenates all of the *`arg`*s with separator spaces, then evaluates the result as a Tcl script and returns its result.

- `for` *`init test reinit body`*

Executes *`init`* as a Tcl script, then evaluates *`test`* as an expression. If it evaluates to true, it executes *`body`* as a Tcl script, executes *`reinit`* as a Tcl script, and reevaluates *`test`* as an expression. Repeats until *`test`* evaluates to false. Returns an empty string.

- `foreach` *`varName list body`*

  `foreach` *`varlist1 list1`* `?`*`varlist2 list2`* `...?` *`body`*

For each element of *`list`*, in order, sets the variable *`varName`* to that value and executes *`body`* as a Tcl script. Returns an empty string. *`list`* must be a valid Tcl list. In the general case, `foreach` can iterate over multiple lists as well as process multiple elements from a list in each iteration.

- `if` *`test1 body1`* `?elseif` *`test2 body2`* `elseif ...? ?else` *`bodyn`*`?`

Evaluates *`test1`* as an expression. If its value is true, it executes *`body1`* as a Tcl script and returns its value. Otherwise it evaluates *`test2`* as an expression; if its value is true, it executes *`body2`* as a script and returns its value. If no test succeeds, it executes *`bodyn`* as a Tcl script and returns its result.

- `source ?-encoding` *`encodingName`*`?` *`fileName`*

Reads the file whose name is `fileName` and evaluates its contents as a Tcl script. Returns the result of the script. Tcl reads the file using the operating system's default character set encoding unless you provide the `-encoding` option.

- `switch ?options? string {pattern body ?pattern body ...?}`

  `switch ?options? string pattern body ?pattern body ...?`

Matches `string` against each `pattern` in order until a match is found, then executes the `body` corresponding to the matching `pattern`. If the last `pattern` is `default`, it matches anything. Returns the result of the `body` executed, or an empty string if no pattern matches. `options` may be `-exact`, `-glob`, `-regexp`, or `--` to indicate the end of options. When using `-regexp` matching, `-matchvar` and `-indexvar` can be used to access matching regular expression subpatterns.

- `while test body`

Evaluates `test` as an expression. If its value is true, it executes `body` as a Tcl script and reevaluates `test`. Repeats until `test` evaluates to false. Returns an empty string.

8.2 The `if` Command

The `if` command evaluates an expression, tests its result, and conditionally executes a script based on the result. For example, consider the following command, which sets variable `x` to `0` if it was previously negative:

```
if {$x < 0} {
    set x 0
}
```

In this case `if` receives two arguments. The first is an expression and the second is a Tcl script. The expression can have any of the forms for expressions described in Chapter 4. The `if` command evaluates the expression and tests the result; if it is true, `if` evaluates the Tcl script. If the value is false, `if` returns without taking any further action.

`if` commands can also include one or more `elseif` clauses with additional tests and scripts, plus a final `else` clause with a script to evaluate if no test succeeds:

```
if {$x < 0} {
    ...
} elseif {$x == 0} {
    ...
} elseif {$x == 1} {
    ...
} else {
    ...
}
```

This command will execute one of the four scripts indicated by ..., depending on the value of `x`. The result of the command will be the result of whichever script is executed. If an `if` command has no `else` clause and none of its tests succeeds, it executes no script and returns an empty string.
Remember that the expressions and scripts for `if` and other control structures are parsed using the same approach as all arguments to all Tcl commands. It is almost always a good idea to enclose the expressions and scripts in braces so that substitutions are deferred until the command is executed. Furthermore, each open brace must be on the same line as the preceding word or else the newline will be treated as a command separator. The following script is parsed as two commands, which results in an error as there are not enough arguments for the `if` command:

```
if {$x < 0}
{
    set x 0
}
```

8.3 The `switch` Command

The `switch` command tests a value against a number of patterns and executes one of several Tcl scripts depending on which pattern matches. The same effect as `switch` can be achieved with an `if` command that has lots of `elseif` clauses, but `switch` provides a more compact way of expressing the structure. Tcl's `switch` command has two forms; here is an example of the first:

```
switch $x {a {incr t1} b {incr t2} c {incr t3}}
```

The first argument to `switch` is the value to be tested (the contents of variable `x` in the example). The second argument is a list containing one or more pairs of elements. The first argument in each pair is a pattern to compare

against the value, and the second is a script to execute if the pattern matches. The `switch` command steps through these pairs in order, comparing the pattern against the value. As soon as it finds a match, it executes the corresponding script and returns the value of that script as its value. If no pattern matches, no script is executed and `switch` returns an empty string. This particular command increments variable `t1` if `x` has the value `a`, `t2` if `x` has the value `b` or `t3` if `x` has the value `c` and does nothing otherwise.
The second form spreads out the patterns and scripts into separate arguments rather than combining them all into one list:

```
switch $x a {incr t1} b {incr t2} c {incr t3}
```

This form has the advantage that you can invoke substitutions on the pattern arguments more easily, but most people prefer the first form because you can easily spread the patterns and scripts across multiple lines like this:

```
switch $x {
    a {incr t1}
    b {incr t2}
    c {incr t3}
}
```

The outer braces keep the newlines from being treated as command separators. With the second form you would have to use backslash-newlines like this:

```
switch $x \
    a {incr t1} \
    b {incr t2} \
    c {incr t3}
```

The `switch` command supports three forms of pattern matching. You can precede the value to test with a switch that selects the form you want: `-exact` selects exact string comparison, `-glob` selects pattern matching as in the `string match` command (see Section 5.10 for details), and `-regexp` selects regular expression matching as described in Section 5.11. The default behavior is `-exact`.
For regular expression matching, you can also provide a `-matchvar` argument followed by a variable name. The `switch` command stores a list in the variable where the first element is the string that matches the entire regular expression, the second element consists of the characters that match the first capturing subpattern, and so on. If the regular expression matches no characters, the value of the match variable is an empty list. The `-indexvar`

option is similar to `-matchvar`, but instead of the actual characters matched, the index variable specified receives a list of character indices referring to the matching substrings. The first element of the index variable is a two-element sublist specifying the indices within the test string of the first and last characters matching the regular expression; the second element is a sublist specifying the indices of the characters matching the first capturing subpattern; and so on.

Note

If the test value starts with a `-` character, the `switch` command can mistake it for an option, causing an error. In general, you should always use the `--` option to mark the end of options and ensure that `switch` correctly identifies the test string in all circumstances.

If the last pattern in a `switch` command is `default`, it matches any value; thus `switch` executes the `default` script if no other patterns match. For example, the following script examines a list and produces three counters. The first, `t1`, counts the number of elements in the list that contain an `a`. The second, `t2`, counts the number of elements that are unsigned decimal integers. The third, `t3`, counts all of the other elements:

```
set t1 0
set t2 0
set t3 0
foreach i $x {
    switch -regexp -- $i {
        a           {incr t1}
        ^[0-9]+$    {incr t2}
        default     {incr t3}
    }
}
```

If a script in a `switch` command is `-`, `switch` uses the script for the next pattern instead. This makes it easy to have several patterns that execute the same script, as in the following example:

```
switch -- $x {
    a -
    b -
    c {incr t1}
    d {incr t2}
}
```

This script increments variable `t1` if `x` is `a`, `b`, or `c`, and it increments `t2` if `x` is `d`.

Note

A common error for newcomers to Tcl is improper placement of comments in a `switch` statement. You can place a comment only where the Tcl interpreter expects to find a Tcl command. In a `switch` statement, that means that you must place comments inside the scripts:

```
switch -- $x {
    # This comment will cause an error
    abc {...}
    ...
}
switch -- $x {
    abc {
          # This comment is okay
          ...
    }
    ...
}
```

8.4 Looping Commands: `while`, `for`, and `foreach`

Tcl provides three commands for looping: `while`, `for`, and `foreach`. Each of these commands executes a script over and over again; they differ in the kinds of setup they do before each iteration and in the ways they decide to terminate the loop.
The `while` command takes two arguments: an expression and a Tcl script. It evaluates the expression and if the result is nonzero, it executes the Tcl script. This process repeats over and over until the expression evaluates to false, at which point the `while` command terminates and returns an empty string. For example, the following script copies a list from variable `a` to variable `b`, reversing the order of the elements along the way:

```
set b {}
set i [expr {[llength $a] - 1}]
while {$i >= 0} {
    lappend b [lindex $a $i]
    incr i -1
}
```

The for command is similar to while except that it provides more explicit loop control. The program to reverse the elements of a list can be rewritten using for as follows:

```
set b {}
for {set i [expr {[llength $a] - 1}]} {$i >= 0} {incr i -1} {
    lappend b [lindex $a $i]
}
```

The first argument to for is an initialization script; the second is an expression that determines when to terminate the loop; the third is a re-initialization script, which is evaluated after each execution of the loop body before the test is evaluated again; and the fourth argument is a script that forms the body of the loop. for executes its first argument (the initialization script) as a Tcl command, then evaluates the expression. If the expression evaluates to true, for executes the body followed by the re-initialization script and reevaluates the expression. It repeats this sequence over and over again until the expression evaluates to false. If the expression evaluates to false on the first test, neither the body script nor the re-initialization script is executed. Like while, for returns an empty string as the result.

for and while are equivalent in that anything you can write using one command you can also write using the other command. However, for has the advantage of placing all of the loop control information in one place where it is easy to see. Of course, in some situations the loop initialization or re-initializaion either is more complex or is nonexistent, and in these cases a while loop may make more sense.

The foreach command iterates over all of the elements of a list. For example, the following script provides yet another implementation of list reversal:

```
set b {}
foreach i $a {
    set b [linsert $b 0 $i]
}
```

The simplest form of foreach takes three arguments. The first is the name of a variable, the second is a list, and the third is a Tcl script that forms the body

of the loop. foreach executes the body script once for each element of the list, in order. Before executing the body in each iteration, foreach sets the variable to hold the next element of the list. Thus, if variable a has the value first second third in the preceding example, the body is executed three times. In the first iteration i has the value first, in the second iteration it has the value second, and in the third iteration it has the value third. At the end of the loop, b has the value third second first and i has the value third. As with the other looping commands, foreach always returns an empty string.
In addition to a single variable name, the foreach command can accept a list of variable names. In this case, each iteration of the loop assigns consecutive element values to the corresponding variable names, so if you provide three variable names, foreach processes the list three elements at a time. The loop iterates until all elements have been used; if a value list doesn't contain enough elements for each loop variable on the last iteration, empty strings are used for the missing elements:

```
foreach {x y} {a b c d e} {
    puts "<$x> <$y>"
}
⇒ <a> <b>
  <c> <d>
  <e> <>
```

The foreach command also can process multiple lists in parallel, with a separate set of variables for each list:

```
foreach i {a b} {j k} {v w x y z} {
    puts "i:<$i>, j:<$j>, k:<$k>"
}
⇒ i:<a>, j:<v>, k:<w>
  i:<b>, j:<x>, k:<y>
  i:<>, j:<z>, k:<>
```

8.5 Loop Control: break and continue

Tcl provides two commands that can be used to abort part or all of a looping command: break and continue. These commands have the same behavior as the corresponding statements in C. Neither takes any arguments. The break command causes the innermost enclosing looping command to terminate immediately. For example, suppose that in the list reversal example in the preceding section we want to stop as soon as an element equal to `ZZZ` is found in the source list. In other words, the result list should consist of a reversal of only those source elements up to (but not including) a `ZZZ` element. This can be accomplished with `break` as follows:

```
set b {}
foreach i $a {
    if {$i == "ZZZ"} break
    set b [linsert $b 0 $i]
}
```

The `continue` command causes only the current iteration of the innermost loop to be terminated; the loop continues with its next iteration. In the case of `while`, this means skipping out of the body and reevaluating the expression that determines when the loop terminates; in `for` loops, the re-initialization script is executed before the termination condition is reevaluated. For example, the following program is another variant of the list reversal example, where `ZZZ` elements are simply skipped without being copied to the result list:

```
set b {}
foreach i $a {
    if {$i == "ZZZ"} continue
    set b [linsert $b 0 $i]
}
```

8.6 The `eval` Command

`eval` is a general-purpose building block for creating and executing Tcl scripts. It accepts any number of arguments, concatenates them with separator spaces, and then executes the result as a Tcl script. All Tcl parsing rules apply to the script, so the script can contain multiple commands, span multiple lines, include comments, and so on.
One use of `eval` is for generating commands, saving them in variables, and then later evaluating the variables as Tcl scripts. For example, the script

```
set reset {
    set a 0
    set b 0
    set c 0
}
...
eval $reset
```

clears variables `a`, `b`, and `c` to `0` when the `eval` command is invoked. In this case, there is no advantage to assigning the script to a variable only to execute it with `eval`; it makes much more sense to execute the three `set` commands directly. But if you are writing an application in which the script is created as a result of dynamic processing, `eval` is an appropriate way to execute the script.

Historically the most important use for `eval` has been to force another level of parsing. The Tcl parser performs only one level of parsing and substitution when parsing a command; the results of one substitution are not reparsed for other substitutions. However, there are times when another level of parsing is necessary, and `eval` provides the mechanism to achieve this.

Most commonly, this situation arises when you have a list of values, stored in either a variable or the return value of a command, and you need to pass the list to a command as separate values. For example, suppose that a variable `vars` contains a list of variables and that you wish to unset each of these variables. One solution is to use the following script:

```
set vars {a b c d}
foreach i $vars {
    unset $i
}
```

This script works just fine, but the `unset` command takes any number of arguments so it should be possible to unset all of the variables with a single command. Unfortunately the following script does not work:

```
set vars {a b c d}
unset $vars
```

The problem with this script is that all of the variable names are passed to `unset` as a single argument, instead of there being a separate argument for each name. Thus `unset` tries to unset a variable named `a b c d`.

As of Tcl 8.5, the preferred solution is to use the `{*}` syntax for argument expansion, as described in Section 2.8:

```
set vars {a b c d}
unset {*}$vars
```

Prior to Tcl 8.5, the only solution was to use `eval`, as with the following command:

```
set vars {a b c d}
eval unset $vars
```

`eval` concatenates its arguments to form a new command `unset a b c d`, which it then passes to Tcl for evaluation. The command string gets reparsed, so each variable name ends up in a different argument to `unset`.

Note

As long as the variable names in this example are provided in a well-formed Tcl list with only space and tab characters as element delimiters, this approach works even if some of the variable names contain spaces or special characters such as `$`. The command

```
eval unset $vars
```

is identical to the command

```
eval [concat unset $vars]
```

In either case, the script evaluated by `eval` is a proper list whose first element is `unset` and whose other elements are the elements of `vars`.

8.7 Executing from Files: `source`

The `source` command reads a file and executes the contents of the file as a Tcl script. `source` takes a single argument that specifies the name of the file. For example, the command

```
source init.tcl
```

executes the contents of the file `init.tcl`. You can specify the file using either an absolute path or a path relative to the present working directory of the

script currently executing.
The return value from `source` is the value returned when the file contents are executed, which is the return value from the last command in the file. In addition, `source` allows the `return` command to be used in the file's script to terminate the processing of the file. See Section 9.2 for more information on `return`.
Using the `source` command, you can break a large script into smaller modules, and then have one main script `source` the other script modules. You can create libraries of reusable procedures by placing the procedure definitions in a file that you can `source` from multiple applications. For information on creating more elaborate script libraries, see Chapter 14.

9. Procedures

A Tcl procedure is a command that you define with a Tcl script. You can define new procedures at any time with the `proc` command described in this chapter. Procedures make it easy for you to package solutions to problems so that they can be reused easily.
Tcl also provides special commands for dealing with variable scopes. Among other things, these commands allow you to pass arguments by reference instead of by value and to implement new Tcl control structures as procedures.

9.1 Commands Presented in This Chapter

The following Tcl commands relate to procedures and variable scoping:

- `proc` *`name argList body`*

Defines a procedure whose name is *`name`*, replacing any existing command by that name. *`argList`* is a list with one element for each of the procedure’s arguments, and *`body`* contains a Tcl script that is the procedure’s body. Returns an empty string.

- `apply` *`{argList body ?namespace?} ?arg1 arg2 ...?`*

Applies the anonymous procedure to the arguments and returns the result. The procedure definition consists of a two- or three-element list. The *`argList`* and *`body`* elements are specified as with `proc`. The optional *`namespace`* element specifies a namespace in which to evaluate the procedure.

- `return` *`?options? ?value?`*

Returns from the innermost nested procedure or `source` command with *`value`* as the result of the procedure. *`value`* defaults to an empty string. Additional options may be used to trigger an exceptional return (see Section 13.5).

- `global` *`name1 ?name2 ...?`*

Binds variable names *`name1`*, *`name2`*, etc. to global variables. References to these names will refer to global variables instead of local variables for the duration of the current procedure. Returns an empty string.

- `upvar` *`?level? otherVar1 myVar1 ?otherVar2 myVar2 ...?`*

Binds the local variable named *`myVar1`* to the variable at stack level *`level`* whose name is *`otherVar1`*. For the duration of the current procedure, variable references to *`myVar1`* will be directed to *`otherVar1`* instead. Additional bindings may be specified with *otherVar2* and *myVar2*, etc. *level* has the same syntax and meaning as uplevel and defaults to 1. Returns an empty string.

- uplevel ?*level*? *arg* ?*arg arg* ...?

Concatenates all of the *arg*s with spaces as separators, then executes the resulting Tcl script in the variable context of stack level *level*. *level* consists of a number or a number preceded by # and defaults to 1. Returns the result of the script.

9.2 Procedure Basics: proc and return

Procedures are created with the proc command, as in the following example:

```
proc plus {a b} { expr {$a+$b} }
```

The first argument to proc is the name of the procedure to create, plus in this case. The second argument is a list of names of arguments to the procedure (a and b in the example). The third argument to proc is a Tcl script that forms the body of the new procedure. After the proc command completes, a new command, plus, exists, which can be invoked just like any other Tcl command. When plus is invoked, Tcl arranges for the procedure's body to be evaluated with the variables a and b set to the values of the arguments. plus must always be invoked with exactly two arguments; the Tcl interpreter raises an error if you invoke a procedure with the wrong number of arguments. The return value for the plus command is the value returned by the last command in plus's body. Here are some correct and incorrect invocations of plus:

```
  plus 3 4
⇒ 7
  plus 3 -1
⇒ 2
  plus 1
∅ wrong # args: should be "plus a b"
```

It is important to realize that proc is just an ordinary Tcl command. It is not a declaration with special syntax, as you might see in other languages such as C. The arguments to proc are processed in the same way as for any other Tcl command. For example, the braces in the argument {a b} are not a special syntactic construct for this command; they are used in the normal fashion to pass both of plus's argument names to proc as a single list of argument names. Technically, if a procedure has only a single argument, the braces aren't

needed around its name, though most Tcl programmers still use them for consistency. Similarly, the braces around `proc`'s last argument are used to pass the entire script body to `proc` as a single argument without performing substitutions on its contents.
If you would like a procedure to return early without executing its entire script, you can invoke the `return` command: it causes the enclosing procedure to return immediately, and the argument to `return` is the result of the procedure. Here is an implementation of a factorial function that uses `return`:

```
  proc fac {x} {
      if {$x <= 1} {
          return 1
      }
      return [expr {$x * [fac [expr {$x-1}]]}]
  }
  fac 4
⇒ 24
  fac 0
⇒ 1
```

If the argument to `fac` is less than or equal to `1`, `fac` invokes `return` to return immediately. Otherwise `fac` executes the `expr` command, which recursively calls `fac` and returns the result.

Note

In this example you could invoke the `expr` command without the "enclosing" `return` command. The `expr` command would be the last command in the procedure's body, so its result would be returned as the result of the procedure. However, using the `return` command in cases like this makes your intention clear to programmers maintaining your code, so it is less likely that someone will add commands to the end of the procedure definition and inadvertently change its behavior.

9.3 Local and Global Variables

When the body of a Tcl procedure is evaluated, it uses a different set of variables from its caller. These variables are called *local variables*, since they are accessible only within the procedure and are deleted when the procedure returns. Variables defined outside of a procedure are called *global variables*. Global variables are persistent, existing until explicitly deleted. Tcl also supports *namespace variables*, which are persistent variables existing in the context of a particular namespace. Chapter 10 discusses namespaces and the use of namespace variables. It is possible to have a local variable with the same name as a global variable, namespace variable, or a local variable in another active procedure, but these are different variables: changes to one do not affect any of the others. If a procedure is invoked recursively, each recursive invocation has a distinct set of local variables.

The arguments to a procedure are just local variables whose values are set from the words of the command that invoked the procedure. When execution begins in a procedure, the only local variables with values are those corresponding to the arguments passed to the procedure. Other local variables are created automatically when they are set.

A procedure can reference global variables with the `global` command. For example, the following command makes the global variables `x` and `y` accessible inside a procedure:

```
global x y
```

The `global` command treats each of its arguments as the name of a global variable and arranges for references to those names within the procedure to be directed to global variables instead of local ones. `global` can be invoked at any time during a procedure; once it has been invoked, it remains in effect until the procedure returns.

Note

Tcl does not provide a form of variables equivalent to static variables in C, which are limited in scope to a given procedure but have values that persist across calls to the procedure. In Tcl you must use global or namespace variables for purposes like this. In general, you should prefer using namespace variables to avoid name conflicts with other such variables.

9.4 Defaults and Variable Numbers of Arguments

In the examples so far, the second argument to `proc` (which describes the arguments to the procedure) has taken a simple form consisting of the names of the arguments. Three additional features are available for specifying arguments. First, the argument list may be specified as an empty string. In this case the procedure takes no arguments, and trying to invoke it with arguments results in an error. For example, the following command defines a procedure that prints out two global variables:

```
proc printVars {} {
    global a b
    puts "a is $a, b is $b"

}
```

The second additional feature is that defaults may be specified for some or all of the arguments. The argument list is actually a list of lists, in which each sublist corresponds to a single argument. If a sublist has only a single element (which has been the case in the previous examples), that element is the name of the argument. If a sublist has two arguments, the first is the argument's name and the second is a default value for it. For example, here is a procedure that increments a given value by a given amount, where the amount defaults to `1`:

```
proc inc {value {increment 1}} {
    expr $value+$increment
}
```

The first element in the argument list, `value`, specifies a name with no default value. The second element specifies an argument with the name `increment` and a default value of `1`. This means that `inc` can be invoked with either one or two arguments:

```
  inc 42 3
⇒ 45
  inc 42
⇒ 43
```

If a default isn't specified for an argument in the `proc` command, the argument must be supplied whenever the procedure is invoked. The defaulted arguments, if any, must be the last arguments for the procedure. This is true both in the `proc` command and when invoking a procedure. If a default is specified for a particular argument, defaults must be provided for all the arguments following that one; similarly, if an argument is omitted when the

procedure is invoked, all the arguments after it must also be omitted.
The third special feature in argument lists is support for variable numbers of arguments. If the last argument in the argument list has the special name args, the procedure may be called with varying numbers of arguments. Arguments before args in the argument list are handled as before, but any number of additional arguments may be specified. The procedure's local variable args is set to a list whose elements are all of the extra arguments. If there are no extra arguments, args is set to an empty string. For example, the following procedure takes any number of arguments and returns their sum:

```
  proc sum {args} {
      set total 0
      foreach val $args {
          set total [expr {$total + $val}]
      }
      return $total
  }
  sum 1 2 3 4 5
⇒ 15
  sum
⇒ 0
```

If a procedure's argument list contains additional arguments before args, they may be defaulted as just described. No default value may be specified for args—the empty string is its default.

9.5 Call by Reference: upvar

Tcl supports call-by-value argument passing only. When you invoke a Tcl command, copies of the argument values are passed to the command. This is true even if the value came from a variable, because the Tcl interpreter substitutes the value of the variable before executing the command. Thus, in the following example, all the sum command receives are copies of the values that are stored in the a and b variables:

```
sum $a $b
```

Tcl does not support true pointer or reference types, either, so it would seem at first impossible to write a procedure that could modify the value of an existing variable. However, the name of a variable is simply a string value, which can in turn be stored in another variable. Therefore, by requesting additional rounds of substitution, we can emulate the behavior of a reference, like so:

```
    set x "The value of x"
    set y x ;# This stores the single character "x" in y
    set $y
⇒ The value of x
```

In this example, the Tcl interpreter substitutes `$y` with its string value, `x`. The `set` command then executes, interprets its argument as the name of a variable, and returns the value stored in the variable. This concept, in combination with a Tcl command called `upvar`, allows us to implement the equivalent of call-by-reference behavior.
The `upvar` command provides a general mechanism for accessing variables outside the context of a procedure. It can be used to access global variables, namespace variables, or local variables in some other active procedure. Most often it is used to implement the equivalent of call-by-reference argument passing, which is particularly useful for arrays. If `a` is an array, you cannot pass it to a procedure `myproc` with a command like `myproc $a`, because there is no value for an array as a whole; there are values only for the individual elements. Instead, you can pass the name of the array to the procedure, as in `myproc a`, and use the `upvar` command to access the array's elements from the procedure.
Here is a simple example of `upvar` in a procedure that prints out the contents of an array:

```
    proc printArray {name} {
        upvar $name a
        foreach el [lsort [array names a]] {
            puts "$el = $a($el)"
        }
    }
    set info(age) 37
    set info(position) "Vice President"
    printArray info
⇒ age = 37
    position = Vice President
```

When `printArray` is invoked, it is given the name of an array as an argument. The `upvar` command then makes this array accessible through the local variable `a` in the procedure. The first argument to `upvar` is the name of a variable accessible to the procedure's caller. This may be a global variable, as in the example, a namespace variable, or a local variable in a calling procedure. The second argument is the name of a local variable. `upvar` arranges things so that accesses to the local variable `a` actually refer to the variable in the caller whose name is given by the variable `name`. In the example this means that when `printArray` reads elements of `a`, it is actually reading elements of the `info` global variable. If `printArray` were to write `a`, it would modify `info`. `printArray` uses the `array names` command to retrieve a list of all the elements in the array. Then it sorts them with `lsort` and prints each of the elements in order.

Note

In the example it appears as if the output is returned as the procedure’s result; in fact, it is printed by the procedure directly to standard output, and the result of the procedure is an empty string.

The first variable name in an `upvar` command by default refers to the context of the current procedure’s caller. However, it is also possible to access variables from any level on the call stack, including the global level. For example,

```
upvar #0 other x
```

makes the global variable `other` accessible via the local variable `x` (the `#0` argument specifies that `other` should be interpreted as a global variable, regardless of how many nested procedure calls are active), and

```
upvar 2 other x
```

makes the variable `other` in the caller of the caller of the current procedure accessible as the local variable `x` (`2` specifies that the context of `other` is two levels up the call stack). The level `0` (as opposed to `#0`) refers to the current context. See the reference documentation for more information on specifying a level in `upvar`.

Note

Although omitting the level argument causes `upvar` to default to the context of the procedure’s caller, the best practice is to explicitly provide a level argument of `1` in this case. This prevents `upvar` from raising an error if the first variable name begins with a `#` or digit.

9.6 Creating New Control Structures: `uplevel`

The `uplevel` command is a cross between `eval` and `upvar`. It evaluates its argument(s) as a script, just like `eval`, but the script is evaluated in the variable context of a different call stack level, like `upvar`. With `uplevel` you can define new control structures as Tcl procedures. For example, here is a new control flow command called `do`:

```
proc do {varName first last body} {
    upvar $varName v
    for {set v $first} {$v <= $last} {incr v} {
        uplevel $body
    }
}
```

The first argument to `do` is the name of a variable. `do` sets that variable to consecutive integer values in the range between its second and third arguments and executes the fourth argument as a Tcl command once for each setting. Given this definition of `do`, the following script creates a list of squares of the first five integers:

```
set squares {}
do i 1 5 {

     lappend squares [expr $i*$i]
  }
  set squares
⇒ 1 4 9 16 25
  set i
⇒ 6
```

The `do` procedure uses `upvar` to access the loop variable (`i` in the example) as its local variable `v`. Then `do` uses the `for` command to increment the loop variable through the desired range. For each value, it invokes `uplevel` to execute the loop body in the variable context of the caller; this causes references to the variables `squares` and `i` in the body of the loop to refer to variables in `do`'s caller. If `eval` were used instead of `uplevel`, `squares` and `i` would be treated as local variables in `do`, which would not produce the desired effect.

Note

This implementation of `do` does not handle exceptional conditions properly. For example, if the body of the loop contains a `return` command, it causes only the `do` procedure to return, which is more like the behavior of `break`. A `return` that occurs in the body of a built-in control flow command such as `for` or `while` causes the procedure that invoked the command to return. In Chapter 13 you will see how to implement this behavior for `do`.

As with `upvar`, `uplevel` takes an optional initial argument that specifies an explicit stack level. In the common case where the script should be evaluated in the context of the caller, the best practice is to explicitly indicate a level of `1`. Otherwise, a script argument beginning with a `#` character or digit would be misinterpreted as the level argument. See the reference documentation for details.

9.7 Applying Anonymous Procedures

Not only do procedures provide a mechanism for modular programming, but they also have performance advantages in Tcl. Internally, the body of a procedure is transformed into an efficient bytecode representation. Procedures also offer a local scope, allowing for the creation of transient variables without potential naming conflicts with existing variables. There are occasions when you might desire the efficiency or encapsulation of a procedure but require the procedure for a single purpose or a limited time. Code callbacks, such as for widget commands and file event handlers, and one-shot transformational functions are typical examples.
The `apply` command, introduced in Tcl 8.5, provides the ability to apply an anonymous function to a set of arguments:

apply *{argList body ?namespace?} ?arg1 arg2 ...?*

The first argument to `apply` is a procedure definition, consisting of a two- or three-element list. The first element is the formal procedure argument, defined in the same way as for `proc`. The second element is a Tcl script implementing the body of the procedure. The third element is optional; if provided, it specifies a namespace in which to evaluate the procedure. (See

Chapter 10 for a discussion of namespaces.) Subsequent arguments to `apply` are the actual values assigned to the procedure arguments.
As an example, consider the following anonymous procedure that calculates the sum of several numbers:

```
    apply { {args} {
        set total 0
            foreach val $args {
            set total [expr {$total + $val}]
        }
        return $total
    } } 1 2 3 4 5 6 7
⇒ 28
    set total
∅ can't read "total": no such variable
```

In this case, the numbers 1 through 7 are assigned as a list to the anonymous procedure's formal argument `args`, just as they would be for a named procedure created with `proc`. The anonymous procedure then calculates and returns the sum of the values. Note that in this example the variable `total` is local to the procedure and is automatically deleted on the procedure's termination.
As a more representative example, consider the task of sorting a list based on the string length of each element. The `lsort` command (discussed in Chapter 6) doesn't have an option for sorting by element length, but you can use the `-command` option to specify your own sorting procedure. The procedure must accept two elements, returning an integer less than, equal to, or greater than zero if the first element is to be considered less than, equal to, or greater than the second, respectively. Although you could define a named procedure to implement the comparison, you can also employ an anonymous procedure:

```
    set states {California Delaware Hawaii Indiana Iowa}
    lsort -command { apply { {e1 e2} {
        expr {[string length $e1] - [string length $e2]}
    } } } $states
⇒ Iowa Hawaii Indiana Delaware California
```

Another use of an anonymous procedure is to implement a callback, such as for variable tracing. (See Chapter 15 for more information on tracing variable access.) Variable trace procedures are called with three values, describing the variable and the type of access. In this example, an anonymous procedure reports the new value of the traced variable on the console:

```
    set vbl "initial"
    trace add variable vbl write {apply {{v1 v2 op} {
        upvar 1 $v1 v
        puts "updated variable to \"$v\""
    }}}
    set vbl 123
⇒ updated variable to "123"
    set vbl abc
⇒ updated variable to "abc"
```

Additionally, `apply` can be used as a building block for implementing various functional programming constructs. You can find several examples of this on the Tcler’s Wiki (http://wiki.tcl.tk). The following demonstrates the implementation of a `map` command, which accepts a list and returns a new list generated by applying a transformation to each element of the original list:

```
    proc map {lambda list} {
        set result {}
        foreach item $list {
            lappend result [apply $lambda $item]
        }
        return $result
    }
    map {x { expr {$x**2} }} {1 2 3 4 5}
⇒ 1 4 9 16 25
    map {x {return [list [string length $x] $x]}} {A BB CCC DDDD}
⇒ {1 A} {2 BB} {3 CCC} {4 DDDD}
```

10. Namespaces

The Tcl interpreter collects all commands and global variables into groups called *namespaces*, so that commands and variables within one namespace don't interfere with commands in another. These namespaces themselves are arranged in a tree, and commands in one namespace can be imported into another. The root of the tree is the *global namespace*, and it contains all commands and variables that are not created explicitly within some other namespace.
Commands and variables may be created within or used from out of any existing namespace. This is done by prefixing the name of the namespace to the command or variable name, separated by the namespace separator, a double colon. The name of the global namespace is an empty string, but a double colon is usually used in-stead as a synonym.
One of the main uses of namespaces is as a mechanism for building a package of related commands. Namespaces provide assistance for this in the form of *ensembles*, which are used to group the public API of a namespace together and present it in the common command and subcommand style.

10.1 Commands Presented in This Chapter

- `namespace children ?`*`namespace`*`? ?`*`pattern`*`?`

Returns a list of all child namespaces of the given namespace, or the current namespace if no *`namespace`* argument is present. If *`pattern`* is specified, it returns only those child namespaces whose (unqualified) name matches the glob pattern.

- `namespace code `*`script`*

Returns *`script`* in a form that, when evaluated, will cause *`script`* to be evaluated in the current namespace. This is ideally suited to callback script generation, because if the script callback has any arguments appended to it at callback time, those arguments will be passed correctly as extra arguments to the command in *`script`* after allowing for namespace handling.

- `namespace current`

Returns the fully qualified name of the current namespace.

- `namespace delete ?`*`namespace`*` ...?`

Deletes each of the named namespaces.

- `namespace ensemble create ?`*`option value ...`*`?`

Creates an ensemble bound to the current namespace and returns the fully qualified name of the ensemble.

- `namespace ensemble configure `*`ensemble`*` ?`*`option`*`? ?`*`value ...`*`?`

Configures the ensemble named *`ensemble`*. If no *`option`* is specified, it returns a dictionary of all options and their values. If *`option`* is present but no *`value`*, it returns the current value of that option. Otherwise there should be a list of options and the values to set those options to, and the result is the empty string.

- `namespace ensemble exists `*`ensemble`*

Returns `1` if *`ensemble`* is an ensemble command, `0` otherwise.

- `namespace eval `*`namespace body`*

Executes a script inside the namespace called *`namespace`*, creating the namespace if it does not already exist. The result of the command is the result of the last command executed in *`body`*.

- `namespace exists `*`namespace`*

Returns `1` if the named namespace exists, `0` otherwise.

- `namespace export ?-clear? ?`*`pattern ...`*`?`

If one or more `string match`–style patterns are specified, appends them to the current namespace's list of command export patterns. If `-clear` is also specified, it clears the list of export patterns first. If no patterns are present, it returns the current list of export patterns.

- `namespace forget ?`*`pattern ...`*`?`

Each command previously imported into the current namespace that matches any of the supplied `string match`-style patterns is "forgotten"; that is, it is deleted from the current namespace. Commands that were not imported and the original exported commands are unaffected.

- `namespace import ?-force? ?`*`pattern ...`*`?`

Imports commands matching one or more `string match`–style patterns into the current namespace. The commands must also match one of their source namespace's export patterns. Each pattern is a qualified glob pattern, consisting of a literal namespace name prefix and a glob pattern suffix. The local name in the current namespace of each imported command will be the same as the local name of the command in the namespace from which it was imported. It is an error to import a command over an existing command other than a previous import of the same command unless the `-force` option is specified, when the existing command is silently replaced instead.

- `namespace origin `*`command`*

Returns the fully qualified name of the command that implements *`command`*. If `command` was imported from another namespace, it returns the qualified name of the real implementation.

- `namespace parent ?namespace?`

Returns the name of the parent namespace of `namespace`, or the parent namespace of the current namespace if no namespace is supplied.

- `namespace path ?list?`

Sets or returns the current namespace's name resolution path. If `list` is present, the name resolution path is set to the list of namespaces named in it. If `list` is absent, the current name resolution path is returned.

- `namespace qualifiers string`

Returns everything up to the last namespace separator in `string`. If `string` does not contain any namespace separators, it returns the empty string. Note that this command does not check whether any namespaces exist.

- `namespace tail string`

Returns everything after the last namespace separator in `string`. If `string` does not contain any namespace separators, it returns `string` unchanged. Note that this command does not check whether any namespaces exist.

- `namespace unknown ?script?`

Returns the current namespace's unknown command handler script, or sets it to `script` if that is supplied. The default unknown command handler for all namespaces is `::unknown`.

- `namespace upvar namespace otherVar localVar \?otherVar localVar ...?`

Maps each of the variables named `otherVar` in the namespace `namespace` to a local variable, `localVar`. The variables are linked; any change to one results in a change to the other.

- `namespace which ?-command|-variable? name`

Returns the fully qualified name of the entity called `name`. If `-command` is given, and by default, `name` is assumed to refer to a command. If `-variable` is given, `name` is assumed to refer to a variable.

- `variable varName ?value? ?varName value ...?`

Creates and refers to, and optionally initializes, variables in the current namespace.

10.2 Evaluating Tcl Code in a Namespace

To execute a script in a namespace, use the `namespace eval` command. This takes the name of a namespace in which to execute and a Tcl script to evaluate in that namespace, and it returns the result of executing the script. It also creates the namespace if it does not already exist. The `proc` command always creates commands relative to the current namespace unless they are fully specified. Note that all commands in the script that are not found in the namespace itself are looked up in the global namespace; for example:

```
    proc whoAmI {} {
        return "global command"
    }
    namespace eval ns {
        proc whoAmI {} {
            return "namespace command"
        }
    }
    whoAmI
 ⇒ global command
    ns::whoAmI
 ⇒ namespace command
    namespace eval ns {
        whoAmI
    }
 ⇒ namespace command
```

You can also declare a procedure within any namespace by giving a fully qualified name to the `proc` command, such as `ns::newCmd`.
To delete a namespace, as opposed to a command within it, use the `namespace delete` command. This takes a list of namespaces to delete.
Access to variables within a namespace is set up using the `variable` command. This takes the name of a variable to set up within the current namespace and an optional value to set the variable to. If evaluated within a procedure, it also arranges for the namespace variable with the given name to be available within the procedure without qualification.

```
  namespace eval counter {
      variable num 0
      proc next {} {
          variable num
          return [incr num]
      }
      proc reset {} {
          variable num
          set num 0
      }
  }
  counter::next
⇒ 1
  counter::next
⇒ 2
  counter::reset
⇒ 0
  counter::next
⇒ 1
```

Note

Always declare variables using the `variable` command. Within a namespace, if you access a variable that hasn't explicitly been declared as a namespace variable, Tcl first checks to see whether there exists a global variable with that name. If an existing global variable is found, Tcl uses that in favor of creating a namespace variable. Although this behavior might seem unexpected at first, it was designed for easy access to predefined global variables such as `argv`, `env`, etc., from within a namespace.

The `variable` command cannot initialize array values, but it can set the variables up within the namespace or grant access to them from a procedure. This means that arrays should be initialized in a separate step.

```
namespace eval catalog {
    variable entries
    array set entries {}
    proc add {item} {

          variable entries
          incr entries($item)
      }
      proc getEntries {} {
          variable entries
          return [lsort [array names entries]]
      }
      proc countInstances {item} {
          variable entries
          return $entries($item)
      }
  }
  catalog::add apple
⇒ 1
  catalog::add orange
⇒ 1
  catalog::add apple
⇒ 2
  catalog::add banana
⇒ 1
  catalog::getEntries
⇒ apple banana orange
  catalog::countInstances apple
⇒ 2
```

When you need to generate a script for handling a callback, such as when working with the `lsort` command's `-command` option, widget `-command` options, or the `after` or `trace` commands discussed later in this book, it is best to generate that script using `namespace code`. This wraps additional code around the script to ensure that it gets evaluated in the current namespace rather than in the namespace in which it was invoked. For example, we could extend the previous example with the ability to save the catalog to a file so that it does not get lost when we quit the interpreter, and we could then set it up so that the catalog gets saved automatically every 10 seconds:

```
namespace eval catalog {
    proc save {file} {
        variable entries
        set f [open $file w]
        puts $f [list array set ::catalog::entries \
            [array get entries]]
        close $f
    }
    proc autocommit {interval file} {
        after $interval [namespace code [list \
            autocommit $interval $file]]
        save $file
    }

    # Save to catalogDB.tcl every 10 seconds
    autocommit 10000 catalogDB.tcl
}
```

10.3 Manipulating Qualified Names

Everything in a namespace can be accessed via its *qualified name*. This consists of the name of the namespace with `::` and the local name of the command, variable, or subnamespace appended. These qualified names may be either *absolute*, being specified with a `::` at the front referring to the global namespace, or *relative* to the current namespace (without the double colon). This is analogous to working with absolute versus relative paths on a file system.

For the purpose of creating qualified paths, the global namespace itself can be referenced with just the `::` prefix. Thus, `::set` refers to the `set` command in the global namespace, and `::env` refers to the global `env` array. The `catalog` namespace from the previous section was created as a direct child of the global namespace. Therefore, in the global namespace you can invoke the `add` procedure using a relative qualified name of `counter::add`. From any other namespace you would need to use the absolute qualified name, `::counter::add`.

To work with such qualified names, Tcl provides a number of tools. To get the namespace part of a qualified name, use the `namespace qualifiers` command. It returns the namespace part of the name. The local part of the qualified name can be obtained using the `namespace tail` command.

```
  namespace qualifiers alpha::beta
⇒ alpha
  namespace qualifiers ::alpha::beta::gamma
⇒ ::alpha ::beta
  namespace qualifiers delta
⇒
  namespace tail alpha ::beta
⇒ beta
  namespace tail ::alpha::beta::gamma
⇒ gamma
  namespace tail delta
⇒ delta
```

To construct a qualified name from parts, just put them together with a literal `::`. Note that if you are holding the namespace name in a variable, you should use one of the alternative forms when substituting the variable because the `::` is otherwise interpreted by the `$` variable substitution.

```
set theNS ::alpha::beta
set theCommand $theNS::gamma
∅ can't read "theNS::gamma": no such variable
   set theCommand ${theNS}::gamma
⇒ ::alpha::beta::gamma
```

10.4 Exporting and Importing Namespace Commands

If a namespace presents a public API of some form, it should be exported from the namespace using the `namespace export` command. This command adjusts and inspects a list of glob patterns associated with the namespace that specify which commands defined within the namespace form its public API.
Given a public API exported by some namespace, you can import commands from that API into another namespace so that you can use them without needing the fully qualified name of the command. This is managed by the `namespace import` command, which takes a list of glob patterns that specify the commands to import (out of the set exported by the other namespace):

```
namespace eval src {
    proc a   {} {return "alpha"}
    proc b   {} {return "beta"}
    proc etc {} {return "..."}
    namespace export a b
}
namespace eval dst {
    namespace import ::src::*
    proc c {} { return "charlie"}
    expr {"[a] [b] [c]"}
}
⇒ alpha beta charlie
```

Note

The global namespace does not export any commands by default, and this namespace is left to the application script to manage by convention. Library packages should not export commands from or import commands into the global namespace.

The `namespace import` command has snapshot semantics; it imports only those commands that are exported at the time when it is called. Also, it does not overwrite any existing commands, including preexisting imports, by default. This behavior can be overridden by the `-force` option. If you wish to remove previously imported commands from a namespace without the risk of deleting other commands you may have created, you should use the `namespace forget` command.

10.5 Inspecting Namespaces

Tcl provides a number of tools for inspecting namespaces. You can discover what the current namespace is with the `namespace current` command, what the name of its parent namespace is with `namespace parent`, and what the child namespaces of the current namespace are with `namespace children` (which takes optional namespace and glob pattern arguments for specifying the namespace to inspect and for constraining the list of returned namespaces respectively):

```
  namespace eval example {
      namespace current
  }
⇒ ::example
  namespace eval example {
      namespace parent
  }
⇒ ::
  namespace eval example {
      namespace eval subNS {
          namespace current
      }
  }
⇒ ::example::subNS
  namespace eval example {
      namespace eval abc1 {}
      namespace eval abc2 {}
      namespace eval abc3 {}
  }
  lsort [namespace children example abc*]
⇒ ::example::abc1 ::example::abc2 ::example::abc3
```

To discover the commands and variables defined in a namespace, use the normal `info` subcommands, and qualify the pattern argument with the name of the namespace that you wish to inspect:

```
  namespace eval example {
      proc eg1 {} {}
      proc eg2 {} {}
  }
  lsort [info command example::*]
⇒ ::example::eg1 ::example::eg2
```

On the other hand, if you know the unqualified name of a command or variable, you can get its fully qualified name using `namespace which`:

```
  namespace eval example {
     proc eg {} {}
       proc env {} {}
       list [namespace which eg] [namespace which set] \
               [namespace which -variable env]
  }
⇒ ::example::eg ::set ::env
```

The current list of export patterns can be obtained by using the `namespace export` command with no arguments.

To discover which namespace originally exported a potentially imported command, use the `namespace origin` command. This command returns the fully qualified name of the command from which the argument command came; if the command was defined in the local namespace, this is just the fully qualified name of the command.

```
namespace eval example {
    proc eg1 {} {}
    proc eg2 {} {}
    namespace export *
}
namespace import example::*
namespace origin eg1
⇒ ::example::eg1
```

Note that `namespace origin` can discover the origin of a command even if it has been renamed:

```
rename eg2 example2
namespace origin example2
⇒ ::example::eg2
```

10.6 Working with Ensemble Commands

A very common system of representing a collection of related commands is as a group of subcommands within an *ensemble*. This is a common pattern within Tcl—for example, the `string`, `namespace`, and `clock` commands all follow it—and it is also common in Tk and many extensions. Tcl provides an ensemble mechanism to allow script programmers to create their own groups of related commands with minimal effort. This is done through the `namespace ensemble` command.

10.6.1 Basic Ensembles

The easiest way to create an ensemble is to use `namespace export` (as described in the preceding section) to publish the commands in a namespace that form the API. Using `namespace ensemble create` without any extra arguments in that namespace creates an ensemble based on the public API with the same fully qualified name as the namespace itself. You can also test whether a command is an ensemble using `namespace ensemble exists`, which returns true if the argument given is the name of a command that is an ensemble:

```
  namespace eval example {
      proc plus {x y} {expr {$x + $y}}
      proc multiply {x y} {expr {$x * $y}}
      proc minus {x y} {expr {$x - $y}}
      namespace export add multiply minus
      namespace ensemble create
  }
⇒ ::example
  example add 2 3
⇒ 5
  example multiply 6.0 7
⇒ 42.0
  namespace ensemble exists example
⇒ 1
  namespace ensemble exists example::add
⇒ 0
```

The subcommands of a namespace may be abbreviated uniquely in use; the ensemble command invokes the correct subcommand or, if the abbreviation is not unique, throws a suitable error:

```
  example mul 3.1 1.3
⇒ 4.03
  example m -4 -5
∅ unknown or ambiguous subcommand "m": should be add, minus, or
  multiply
```

Note that procedures placed inside an ensemble generate an ensemble-aware error message when they are invoked with an incorrect number of arguments:

```
  example mul 42
∅ wrong # args: should be "example mul x y"
```

10.6.2 Placing Ensembles Inside Ensembles

You can nest ensembles inside ensembles. This is useful when the ensemble is modeling a complex API, which is fairly common in situations like Tk widgets. In the following example, the `compute` command has two general subcommands doing computations of the area of shapes and the volume of solids. These subcommands in turn have their own specialized sub-subcommands for each of the types of shapes or solids that they support:

```
namespace eval compute {
    variable pi 3.1415927
    namespace export *
    namespace ensemble create
    namespace eval area {
        namespace export *
        namespace ensemble create
    }
    namespace eval volume {
        namespace export *
        namespace ensemble create
    }
}
proc compute::area::circle {radius} {
    variable ::compute::pi
    return [expr {$pi * $radius**2}]
}
proc compute::area::square {side} {
    return [expr {$side ** 2}]
}
proc compute::volume::sphere {radius} {
    variable ::compute::pi
    return [expr {4 * $pi / 3 * $radius**3}]
}
proc compute::volume::cube {edge} {
    return [expr {$edge ** 3}]
}
compute ?
∅ unknown or ambiguous subcommand "?": should be area, or volume
compute area ?
∅ unknown or ambiguous subcommand "?": should be circle, or square
compute area circle 4
⇒ 50.2654832
compute vol sphere 2
⇒ 33.5103221333
```

10.6.3 Controlling the Ensemble Configuration

The `namespace ensemble` command `create` takes a number of option-value pairs, most of which can also be controlled by using the `namespace ensemble configure` subcommand. You can define the list of subcommands explicitly using the `-subcommands` option. This is useful in situations where you are not defining an ensemble to have exactly the publicly exported commands of a namespace, such as when using the global namespace. You can also specify whether unambiguous prefixes of commands are permitted by using the `-prefix` option (true by default), or what the command created by the `namespace ensemble create` subcommand is by using the `-command` option. The namespace that the ensemble is bound to is readable by fetching the `-namespace` option, though this

cannot be set and is determined automatically at creation time.
There are two other configurable options: `-map` and `-unknown`. The `-map` option contains a dictionary that maps subcommand names to lists of words. Each of these lists gives a replacement for the first two words of the command (that is, the ensemble command name and the subcommand name), and the overall command created is executed to implement the subcommand being executed. This provides a general rewriting scheme that allows additional arguments to be passed to any subcommand implementation, which can be extremely useful.

```
  proc raisePower {power value} {
      return "$value ^ $power =\
              [expr {$value ** $power}]"
  }
  namespace ensemble create -command power -map {
      square {raisePower 2}
      cube   {raisePower 3}
      sqroot {raisePower 0.5}
      invert {raisePower -1}
  }
  power cube 4.0
⇒ 4.0 ^ 3 = 64.0
  power sqroot 49
⇒ 49 ^ 0.5 = 7.0
  power inv 0.0625
⇒ 0.0625 ^ -1 = 16.0
```

As you can see, the `-map` option allows for the construction of very complex behavior with a minimum of effort.

10.6.4 Handling Unknown Ensemble Subcommands

The `-unknown` option is a list of words that forms a command prefix that is used whenever a subcommand cannot be resolved to one of the existing subcommands of an ensemble. When an unknown subcommand is encountered, all the arguments to the ensemble (including the name of the ensemble itself) are appended to the command prefix (after proper quoting), and the resulting command is executed. If the returned list is empty, the ensemble reparses the subcommand name and dispatches to the command that now implements the subcommand (that is, if the unknown handler updated at least one of the `-map` or `-subcommand` options of the ensemble) or generates an error. If the returned list is non-empty, it consists of the full list of arguments (including the command name) to execute for the subcommand,

which allows for one-off mappings.
The `-unknown` option is useful when you are creating an overloading of one of the core Tcl commands, as it allows you to do so without having to enumerate the whole list of subcommands that the core Tcl command supports. For example, the `string` command does not have a subcommand to reverse a string, but it is easy to add one like this:

```
rename string strCore
proc strReverse {string} {
    set result {}
    for {set i [strCore length $string]} {$i>0} {} {
        incr i -1
        append result [strCore index $i]
    }
    return $result
}
proc unknownStrCmd {string subcommand args} {
    puts "passing $subcommand through to strCore"
    return [list strCore $subcommand {expand}$args]
}
namespace ensemble create -command string -map {
    reverse {strReverse}
    repeat  {strCore repeat}
    replace {strCore replace}
} -unknown unknownStrCmd
```

All the unknown handler (`unknownStrCmd`) does is write an informative message and hand subcommands through to the (renamed) core version of the `string` command. Note that the `repeat` and `replace` subcommands have been explicitly added to the ensemble subcommand map so that unambiguous subcommand prefixing works as expected.

```
  string reverse abc
⇒ passing length through to strCore
  passing index through to strCore
  passing index through to strCore
  passing index through to strCore
  cba
```

For performance reasons, it is often better for unknown handlers to update the ensemble with the subcommand mapping, like this:

```
proc unknownStrCmd {string subcommand args} {
    catch {strCore $subcommand} msg
    if {![string match {*: must be *} $msg]} {
        puts "adding $subcommand to ensemble"
        set map [namespace ensemble \
                configure $string -map]
           dict set map $subcommand [list \
                strCore $subcommand]
           namespace ensemble configure $string -map $map
       }
       puts "passing $subcommand through to strCore"
       return [list strCore $subcommand {expand}$args]
  }
  string reverse abcdef
⇒ adding length to ensemble
  passing length through to strCore
  adding index to ensemble
  passing index through to strCore
  fedcba
```

Note that the messages about the `length` and `index` subcommands are printed only the first time they are used in the ensemble. After that, the ensemble contains a dynamically built cache of the mapping.

10.7 Accessing Variables from Other Namespaces

Sometimes you need to access variables from a namespace other than the current one, either because the variable represents some shared state, or because it has some special properties. There are several ways to do this.
First, you can use the fully qualified name of the variable. This is useful mainly when the variable is accessed from one place in the code or when it is a special variable such as the `::env` array.
Second, you can use the fully qualified name with `global`, `upvar`, or `variable` to import the variable into the current namespace. This is often used when only a limited number of variables are required and their names are fixed.
Third, you can use `namespace upvar` to import a group of variables from a single namespace. This is particularly useful when that namespace represents a context for some operation and the names of the variables within that potentially variable namespace are static.

```
proc blob {name a b c} {
    namespace eval $name {}
    namespace upvar $name a va b vb c vc
    set va $a
    set vb $b
    set vc $c
    namespace ensemble create -map [list \
        set [list ::blobSet $name] \
        sum [list ::blobSum $name] \
        end [list ::blobEnd $name] \

     ] -command $name
  }
  proc blobSet {ns var value} {
      # more elegant than: upvar #0 ${ns}::${var} v
      namespace upvar $ns $var v
      set v $value
      return
  }
  proc blobSum {ns} {
      namespace upvar $ns  a va  b vb  c vc
      return [expr {$va + $vb + $vc}]
  }
  proc blobEnd {ns} {
      rename $ns {}
      namespace delete $ns
  }
  # Now for some example usage...
  blob example 1 2 3
⇒ ::example
  example set a 4
  example sum
⇒ 9
  example set b 5
  example sum
⇒ 12
  example end
```

10.8 Controlling the Name Resolution Path

When a command is invoked within a namespace, the command is looked up from the name in a process known as *resolution*. Names starting with the namespace separator `::` are always looked up from the global namespace, but names containing a namespace separator that is not at the front of the name are resolved relative to either the current namespace or, if that fails, relative to the global namespace.
When a name does not contain a namespace separator at all, it is resolved using the namespace's *resolution path*. This path is a list of namespaces that is controlled using the `namespace path` command, and it always behaves as if

the current namespace is at the front of the list and the global namespace is on the end. The command is looked up by checking each namespace on the path in order to see if the command is defined there. The `namespace path` command sets the path to the given list or returns the current path if no list is given.
For example, this allows a service namespace to be spliced in, which allows result generation to be done using the `puts` command. This makes going from interactive testing to deployment extremely easy.

```
namespace eval ::collector {
    proc collect {command args} {
        variable accumulator {}
        set ns [namespace parent \
                [namespace origin $command]]
        namespace eval $ns {
            namespace path ::collector
        }
        $command {expand}$args
        namespace eval $ns {
            namespace path {}
        }
        append accumulator "\[generated \
            [string length $accumulator]\
                characters\]"
        return [string trimright $accumulator \n]
    }
    proc puts {args} {
        if {[llength $args] == 1} {
            variable accumulator
            append accumulator [lindex $args 0] \n
            return
        }
        ::puts {expand}$args
    }
}
namespace eval ::example {
    proc makeMessage {name age} {
        puts "Hello $name"
        puts "You are $age years old"
    }
}
collector::collect ::example::makeMessage \
    "Joe Bloggs" "37 and three quarters"
⇒ Hello Joe Bloggs
  You are 37 and three quarters years old
  [generated 57 characters]
```

The `namespace path` command can also be used to make a namespace system that functions very much like objects and classes:

```
set counter 0
proc new {class} {
    global counter
    set cmd ::obj[incr counter]
    namespace eval $cmd [list namespace path ::$class]
    proc $cmd {args} "namespace eval $cmd \$args"
    return $cmd
}
```

This works by creating a new namespace for every command and setting that namespace up to resolve its commands using the class namespace before the global namespace. It then creates a command that just evaluates its arguments in the namespace, which has the effect of resolving the subcommand in the class as there are no commands in the object namespace. The class method procedures should use the `upvar 1` command to access the object namespace; the original caller's stack frame can be reached using the `upvar 2` command, and the class's namespace should be accessed using the `variable` command:

```
  namespace eval example {
      proc foo {args} {puts foo:$args}
      proc bar {args} {puts bar:$args}
      proc set {var args} {
          upvar 1 $var v
          set v {expand}$args
      }
      proc setClass {var args} {
          variable $var
          set $var {expand}$args
      }
  }
  set o1 [new example]
⇒ ::obj1
  $o1 foo a b c
⇒ foo:a b c
```

Notice that it is easy to work with variables in the object, just by using normal Tcl commands with the object name prefixed:

```
  $o1 set x 0
⇒ 0
```

Also note that the preceding variable is created in the object's namespace and not in the class's namespace. The variables in the class are unrelated unless explicitly synchronized.

```
  incr ::o1::x
  $o1 set x
⇒ 1
  set ::example::x "In the class!"
  $o1 set x
⇒ 1
  $o1 setClass x
⇒ In the class!
```

11. Accessing Files

This chapter describes Tcl's commands for dealing with files and directories. The commands allow you to read and write files sequentially or in a random-access fashion. They also allow you to retrieve information kept by the system about files, such as the time of last access. You can rename, copy, and delete files. Last, these commands can be used to manipulate file names; for example, you can remove the extension from a file name or find the names of all files that match a particular pattern.

11.1 Commands Presented in This Chapter

This chapter describes the following commands for manipulating files and directories. Most of these commands also work with interprocess pipes and network sockets, covered in Chapter 12. In Tcl terminology, files, pipes, and network sockets are considered *channels*. All of the I/O-related commands work on any channel.

- `cd ?`*`dirName`*`?`

Changes the current working directory to *`dirName`*, or to the home directory (as given by the `HOME` environment variable) if *`dirName`* isn't given. Returns an empty string.

- `glob ?`*`option ...`*`? ?--? `*`pattern`*` ?`*`pattern ...`*`?`

Returns a list of the names of all files that match any of the *`pattern`* arguments (special characters `?`, `*`, `[]`, `{}`, and `\`). The optional `--` argument signals the end of options. See the reference documentation for more information on allowed options.

- `pwd`

Returns the full path name of the current working directory.

- `chan close `*`channelID`*

  `close `*`channelID`*

Closes the channel given by *`channelID`*. Returns an empty string. The `chan close` command, introduced in Tcl 8.5, simply incorporates the `close` functionality into the `chan` ensemble.

- `chan eof `*`channelID`*

  `eof `*`channelID`*

Returns `1` if an end-of-file condition has occurred during the most recent read from `channelID`, `0` otherwise. Starting with Tcl 8.5, `eof` is deprecated in favor of `chan eof`.

- `file option name ?arg arg ...?`

Performs one of several operations on the file name given by `name` or on the file to which it refers, depending on `option`. See this chapter and the reference documentation for more information.

- `chan flush channelID`

  `flush channelID`

Writes out any buffered output that has been generated for `channelID`. Returns an empty string. Starting with Tcl 8.5, `flush` is deprecated in favor of `chan flush`.

- `chan gets channelID ?varName?`

  `gets channelID ?varName?`

Reads the next line from `channelID` and discards its terminating end-of-line character(s). If `varName` is specified, it places the line in that variable and returns a count of characters in the line (or `-1` for end of file). If `varName` isn't specified, it returns the line of characters read (or an empty string for end of file). The `chan gets` command, introduced in Tcl 8.5, simply incorporates the `gets` functionality into the `chan` ensemble.

- `open name ?access?`

Opens the file `name` in the mode given by `access`. `access` may be `r`, `r+`, `w`, `w+`, `a`, or `a+` or a list of flags such as `RDONLY`; it defaults to `r`. Returns a file identifier for use in other commands like `gets` and `close`. If the first character of `name` is `|`, a command pipeline is invoked instead of a file being opened (see Section 12.4 for more information).

- `chan puts ?-nonewline? ?channelID?string`

  `puts ?-nonewline? ?channelID?string`

Writes `string` to `channelID`, appending a newline character unless `-nonewline` is specified. `channelID` defaults to `stdout`. Returns an empty string. The `chan puts` command, introduced in Tcl 8.5, simply incorporates the `puts` functionality into the `chan` ensemble.

- `chan read ?-nonewline? channelID`

  `chan read channelID numChars`

  `read ?-nonewline? channelID`

  `read channelID numChars`

Reads and returns the next `numChars` characters from `channelID` (or up to the end of the file, if fewer than `numChars` characters are left). If `numChars` is omitted, it reads and returns the remaining characters of the file; when combined with `-nonewline`, the final newline, if any, is dropped. The `chan read` command, introduced in Tcl 8.5, simply incorporates the `read` functionality into the `chan` ensemble.

- `chan seek` *`channelID offset`* `?`*`origin`*`?`

  `seek` *`channelID offset`* `?`*`origin`*`?`

Positions *`channelID`* so that the next access starts at *`offset`* bytes from *`origin`*. *`origin`* may be `start`, `current`, or `end` and defaults to `start`. Returns an empty string. Starting with Tcl 8.5, `seek` is deprecated in favor of `chan seek`.

- `chan tell` *`channelID`*

  `tell` *`channelID`*

Returns the byte offset from the start of the file of the current access position for *`channelID`*. Starting with Tcl 8.5, `tell` is deprecated in favor of `chan tell`.

- `chan configure` *`channelID`* `?`*`optionName`*`? ?`*`value`*`?`

  `?`*`optionName value`* `...?`

  `fconfigure` *`channelID`* `?`*`optionName`*`? ?`*`value`*`?`

  `?`*`optionName value`* `...?`

Queries or sets the configuration options of the channel *`channelID`*. If no *`optionName`* or *`value`* arguments are supplied, the command returns a dictionary of all configuration options and their values. If *`optionName`* is supplied but no *`value`*, the command returns the current *`value`* of the given option. If one or more pairs of *`optionName`* and *`value`* are supplied, the command sets each of the named options to the corresponding value; in this case the return value is an empty string. See the text and the reference documentation for a description of the supported options. Starting with Tcl 8.5, `fconfigure` is deprecated in favor of `chan configure`.

- `chan copy` *`inputChan outputChan`* `?-size` *`size`*`?`

  `?-command` *`callback`*`?`

  `fcopy` *`inputChan outputChan`* `?-size` *`size`*`?`

  `?-command` *`callback`*`?`

Copies data from the channel *`inputChan`*, which must have been opened for reading, to the channel *`outputChan`*, which must have been opened for writing. The `chan copy` command transfers data from *`inputChan`* until either the end of file or the maximum of *`size`* bytes has been transferred, if specified. By default, the command blocks transfer takes place in the foreground, and it returns the number of bytes written to *`outputChan`*. If you provide the `-command` option, `chan copy` returns immediately, and once the copy is completed, *`callback`* is invoked with one or two additional arguments that indicate how many bytes were written to *`outputChan`*. If an error occurred during the background copy, the second argument is the error string associated with the error. Starting with Tcl 8.5, `fcopy` is deprecated in favor of `chan copy`.

11.2 Manipulating File and Directory Names

Every file has a name. File names are specified in Tcl using the normal Unix syntax. For example, the file name `x/y/z` refers to a file named `z` that is located in a directory named `y`, which in turn is located in a directory named `x`, which must be in the current working directory. The file name `/top` refers to a file `top` in the root directory. You can also use tilde notation to specify a file name relative to a particular user's home directory. For example, the name `~ouster/mbox` refers to a file named `mbox` in the home directory of user `ouster`, and `~/mbox` refers to a file named `mbox` in the home directory of the user running the Tcl script. These conventions (and the availability of tilde notation in particular) apply to all Tcl commands that take file names as arguments.

Windows systems differ from these Unix conventions. For Windows systems, a backslash separates directories, rather than a forward slash. The backslash, `\`, can be problematic in Tcl and other languages based on C programming conventions. You normally need to escape a backslash with a second backslash in a context where Tcl can perform substitutions on the path (for example, `\\apps\tcl`), or else protect the path from substitution (for example, `{\apps\tcl}`). This gets more complicated with network shares, when there are two backslashes that must be escaped, as in `\\\\host\\share`. In addition, Windows disk drives do not get mounted under one root directory (as on Unix, Linux, and Mac OS X platforms). Instead, Windows uses a drive letter, such as `C:`, to indicate the disk drive.

To handle the file-naming differences, use the Tcl `file join` and `file split` commands to properly combine file names with directory names in a cross-platform manner. `file` is a general-purpose command with many options that can be used both to manipulate file names and to retrieve information about files. The commands in this section operate purely on file names. They make no system calls and do not check to see if the names correspond to actual files.

Consider the following example, which generates a full path name to a file relative to the user's home directory:

```
  set env(HOME)
⇒ C:\Documents and Settings\Owner
  file join $env(HOME) lib mylib.tcl
⇒ C:/Documents and Settings/Owner/lib/mylib.tcl
```

Notice that the `file join` command returns a Unix-style path that contains `/` characters for directory delimiters, no matter the original format of the individual path components. You can use a "Tcl-native" path format like this with all Tcl commands that access the file system, as well as paths with the platform-specific directory delimiter. In addition, if any component starts at the root level, such as with a leading `/`, the `file join` command discards all the previous components.

Note

Use the `file join` command so that your scripts are not inadvertently tied to just one platform, such as Unix or Windows.

Note that some directory names include spaces. Tcl commands use spaces as separators, so you need to properly group the arguments to the `file join` command; for example:

```
  file join c:/ {Program Files} Tcl tclsh.exe
⇒ c:/Program Files/Tcl/tclsh.exe
```

Use the `file split` command to break a file name path into its component parts; for example:

```
  file split x/y/z
⇒ x y z
```

The `file nativename` command returns a file name formatted for native usage. This is especially useful when you call the `exec` command, covered in Chapter 12, or when writing the path to disk or presenting the path to the user. With the `exec` command, you need to pass file names in the expected, or native, format. For example, the `file nativename` command expands the tilde character to identify a person's home directory:

```
  file nativename ~ericfj/scripts/coolscript.tcl
⇒ /Users/ericfj/scripts/coolscript.tcl
```

`file dirname` removes the last component from a file name, which ostensibly produces the name of the directory containing the file:

```
  file dirname /a/b/c
⇒ /a/b
  file dirname main.c
⇒ .
```

However, the `file` command does not check to be sure that there really is a file or directory corresponding to the name.

`file extension` returns the extension for a file name (all the characters starting with the last `.` in the name), or an empty string if the name contains no extension:

```
  file extension src/main.c
⇒ .c
```

`file rootname` returns everything in a file name except the extension:

```
  file rootname src/main.c
⇒ src/main
  file rootname foo
⇒ foo
```

`file tail` returns the last component of a file name (that is, the name of the file within its directory):

```
  file tail /a/b/c
⇒ c
  file tail foo.txt
⇒ foo.txt
```

The `file normalize` command returns a unique normalized path representation for the file or directory, whose string value can be used as a unique identifier for it. A normalized path is an absolute path that has all `../` and `./` removed. Additionally, on Unix and Mac OS systems the segments leading up to the path must be free of symbolic links/aliases, and on Windows systems it provides the long form with that form's case dependence; for example:

```
  file normalize ~/Documents/../tcltk/intro/examples
⇒ /Users/ken/tcltk/intro/examples
```

The `file pathtype` command tells you information about a file name, such as whether it has an absolute or a relative path. This command returns either `absolute`, `relative`, or `volumerelative`, depending on the type of the file name passed to the command. Relative files have names relative to the current working directory. Similarly, `volumerelative` indicates a name relative to a given mounted volume or disk. You will find many `volumerelative` files on Windows systems. Use the following commands as a guide to `file pathtype`:

```
  file pathtype foo
⇒ relative
  file pathtype [pwd]
⇒ absolute
```

Finally, the `file volumes` command lists all mounted volumes. On Unix and Mac OS X systems, it should return `/` for the root directory. On Windows, it returns a Tcl version of all mounted drives, such as `c:/`.

11.3 The Current Working Directory

With Tcl, you can provide file names that either are absolute paths or are named relative to the current working directory. Thus, it is often important to know which directory is the current working directory.
Tcl provides two commands that help to manage the current working directory: `pwd` and `cd`. `pwd` takes no arguments and returns the full path name of the current working directory. `cd` takes a single argument and changes the current working directory to the value of that argument. If `cd` is invoked with no arguments, it changes the current working directory to the home directory of the user running the Tcl script (`cd` uses the value of the `HOME` environment variable as the path name of the home directory).

11.4 Listing Directory Contents

The `glob` command takes one or more patterns as arguments and returns a list of all the file names that match the pattern(s):

```
   glob *.c *.h
⇒ main.c hash.c hash.h
```

`glob` uses the matching rules of the `string match` command (see Chapter 5). In the preceding example `glob` returns the names of all files in the current directory that end in `.c` or `.h`. `glob` also allows patterns to contain comma-separated lists of alternatives between braces, as in the following example:

```
   glob {{src,backup}/*.[ch]}
⇒ src/main.c src/hash.c src/hash.h backup/hash.c
```

`glob` treats this pattern as if it were actually multiple patterns, each containing one of the strings, as in the following example:

```
glob {src/*.[ch]} {backup/*.[ch]}
```

Note

> The extra braces around the patterns in these examples are needed to keep the brackets inside the patterns from triggering command substitution. They are removed by the Tcl parser in the usual fashion before the command procedure for `glob`.

If a `glob` pattern ends in a slash, it matches only directory names. For example, the command

```
glob */
```

returns a list of all the subdirectories of the current directory.
You can specify that the `glob` command return only particular types of files with the `-types` option. As an argument to `-types`, provide a list of one or more of the following: `b` for block-special devices, `c` for character devices, `d` for directory, `f` for plain file (not directory), `l` ("ell") for symbolic links, `p` for named pipes, and `s` for sockets. You can also provide flags for file access permissions—`r` for read permission, `w` for write permission, `x` for execute permission—as well as `hidden` for hidden files and `readonly` for files with only read permission. You can combine more than one item, but any name returned by `glob` needs to match at least one of the file type items and all of the file permission items. For example, to find all files (but not directories) for which you have both read and write permission, use a command like the following:

```
 glob -types { f r w } *
⇒ b.txt c.txt
```

If you have just one type of item to look for, you can use a simpler syntax:

```
 glob -types f *
⇒ b.txt c.txt
```

The `-directory` option tells `glob` to start in a given directory; for example:

```
 glob -directory /usr/local -types d *
⇒ /usr/local/bin /usr/local/lib
```

In this case, the `*` passed to `glob` gets interpreted inside the given directory, `/usr/local`. This option is useful if the directory name contains characters that would otherwise be treated as wildcards or other metacharacters in the `glob`

pattern.
The `-path` option tells the `glob` command to search for files with names that start with the value given to `-path`. This is mostly useful when you have file names that somehow interfere with the `glob` command. You cannot use the `-path` option with the `-directory` option. The main difference is that the `-directory` option tells `glob` to start in a given directory, whereas `-path` tells `glob` to use names starting with the given prefix. The value passed to `-path` can include a directory name as well as characters that start file names. The `-directory` option accepts only the name of a directory.
The `-tails` option tells the `glob` command to return only the part of the names that comes after the values specified in the `-directory` or `-path` options. For example, compare the following:

```
  glob -directory /usr/local -types d *
⇒ /usr/local/bin /usr/local/lib
  glob -directory /usr/local -types d -tails *
⇒ bin lib
```

The special `-join` option tells `glob` to treat all the remaining options as one large pattern, using a directory separator to join the elements. This basically works like the `file join` command but uses the results as the `glob` path.
By default, `glob` raises an error if no files match the patterns you provide. You can suppress the error condition by supplying the `-nocomplain` option to `glob`, in which case it simply returns an empty list if no files match.

11.5 Working with Files on Disk

Tcl provides a variety of commands for performing operations such as moving, renaming, copying, and deleting files on disk. These commands directly invoke operating system function calls to perform their actions, so they are efficient as well as platform-independent. Your scripts should always use these Tcl commands to perform these actions rather than using `exec` to execute platform-specific commands.

11.5.1 Creating Directories

Create new directories with the `file mkdir` command, short for “make directory” (and named after a Unix and DOS command); for example:

```
    file mkdir foo
    cd foo
    pwd
⇒ /Users/ericfj/foo
```

11.5.2 Deleting Files

Delete files with the `file delete` command:

```
file delete a.txt
```

You must list each file as a separate argument, such as

```
file delete a.txt b.txt
```

This command deletes two files. The `file delete` command does not perform wildcard expansion, so the following example would delete only a file named `*.tmp`:

```
file delete *.tmp
```

The proper way to handle such a case is to use the `glob` command to return a list of files matching the pattern, then use Tcl's argument expansion syntax to provide the list elements as separate arguments to `file delete`:

```
file delete {*}[glob *.tmp]
```

You can use `file delete` to delete a directory simply by providing the directory's name as an argument. However, `file delete` raises an error if the directory is not empty:

```
    file delete foo
∅ error deleting "foo": directory not empty
```

Use the `-force` option to delete the non-empty directory. This option also recursively deletes all files and directories stored under the deleted directory, so use it with care.

11.5.3 Copying Files

While you can write Tcl procedures to read in the contents of a file and write the output to another file, it is far easier to use the Tcl `file copy` command to copy files. The `file copy` command copies one file, the source, to another, the destination; for example:

```
  glob *.txt
⇒ a.txt
  file copy a.txt b.txt
  glob *.txt
⇒ a.txt b.txt
```

The `file copy` command copies the file `a.txt` to `b.txt`. The `glob` command then verifies that the new file, `b.txt`, now exists.
The `file copy` command raises an error if the target file already exists, as in the following example:

```
  file copy a.txt b.txt
∅ error copying "a.txt" to "b.txt": file already exists
```

You can tell the `file copy` command to overwrite the target file with the `-force` option:

```
file copy -force a.txt b.txt
```

You can also copy a number of files to a target directory. For example:

```
file copy a.txt b.txt Documents
```

This command copies two files, `a.txt` and `b.txt`, to the target directory, `Documents`. In this case, the target must be a directory, not a regular file.

11.5.4 Renaming and Moving Files

Use the `file rename` command to give a new name to a file or directory. You need to provide the name of the source file or directory as well as the new name; for example:

```
file rename b.txt c.txt
```

This command renames `b.txt` to its rather original new name, `c.txt`. If the target already exists, you need to provide the `-force` option to cause `file rename` to overwrite the target; otherwise it raises an error.
If the target name refers to another directory, the `file rename` command moves the file to its new location. You can also specify a directory as a source, in

which case the directory is either renamed or moved.

11.5.5 File Information Commands

In addition to the options already discussed, the `file` command provides many other options that can be used to retrieve information about files. Each of these options except `stat` and `lstat` has the form

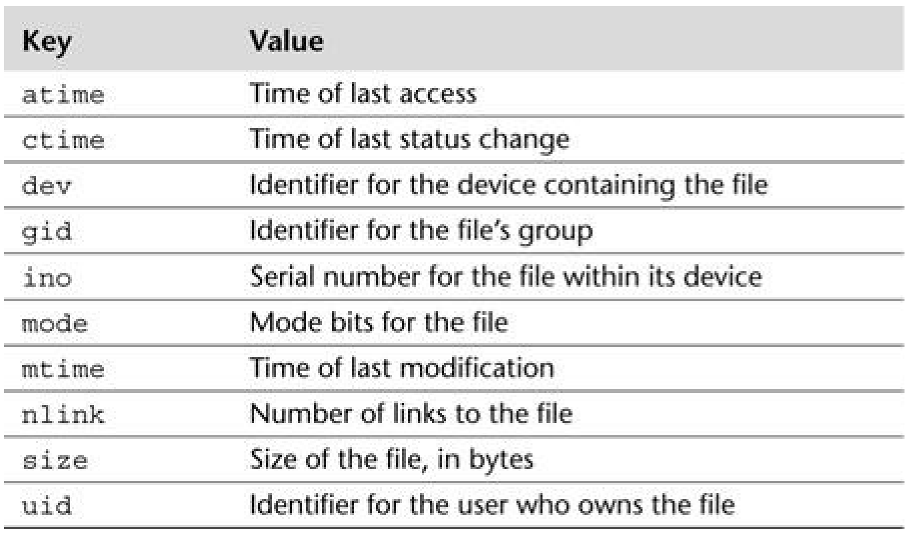
```
file option name
```

where *`option`* specifies the information desired, such as `exists` or `readable` or `size`, and *`name`* is the name of the file.
The `exists`, `isfile`, `isdirectory`, and `type` options return information about the nature of a file. `file exists` returns `1` if a file by the given name exists and `0` if there is no such file or the current user doesn't have search permission for the directories leading to it. `file isfile` returns `1` if the file is an ordinary disk file and `0` if it is something else, such as a directory or device file. `file isdirectory` returns `1` if the file is a directory and `0` otherwise. `file type` returns a string such as `file`, `directory`, or `socket` that identifies the file type.
The `readable`, `writable`, and `executable` options return `0` or `1` to indicate whether the file exists, and if so whether the current user is permitted to carry out the indicated action on the file. The `owned` option returns `1` if the current user is the file's owner and `0` otherwise.
The `size` option returns a decimal string giving the size of the file in bytes. `filemtime` returns the time when the file was last modified. The time value is returned in the standard POSIX form for times, namely, a signed integer giving the number of seconds since January 1, 1970, 00:00 UTC, otherwise known as the *epoch time*. The `atime` option is similar to `mtime` except that it returns the time when the file was last accessed. See Chapter 15 for more information on Tcl's facilities for manipulating epoch time.
The `stat` option provides a simple way to get many pieces of information about a file at one time. Using `stat` can be significantly faster than invoking `file` many times to get the pieces of information individually. `file stat` also provides additional information that isn't accessible with any other file options. It takes two additional arguments, which are the name of a file and the name of a variable, as in the following example:

```
file stat main.c info
```

In this case the name of the file is `main.c` and the variable name is `info`. The variable is treated as an array, and the elements shown in Table 11.1 are set, each as a decimal string.

Table 11.1 Array Elements Returned by the `file stat` Command



Key	Value
`atime`	Time of last access
`ctime`	Time of last status change
`dev`	Identifier for the device containing the file
`gid`	Identifier for the file's group
`ino`	Serial number for the file within its device
`mode`	Mode bits for the file
`mtime`	Time of last modification
`nlink`	Number of links to the file
`size`	Size of the file, in bytes
`uid`	Identifier for the user who owns the file

The `atime`, `mtime`, and `size` elements have the same values as produced by the corresponding `file` options discussed earlier. For more information on the other elements, refer to your system documentation for the `stat` system call; each of the elements is taken directly from the corresponding field of the structure returned by `stat`.
The `lstat` and `readlink` options are useful when dealing with symbolic links, and they can be used only on systems that support symbolic links. `file lstat` is identical to `file stat` for ordinary files, but when it is applied to a symbolic link, it returns information about the symbolic link itself, whereas `file stat` returns information about the file to which the link points. `file readlink` returns the contents of a symbolic link, that is, the name of the file to which it refers; it may be used only on symbolic links. For all of the other `file` commands, if the name refers to a symbolic link, the command operates on the target of the link, not the link itself.

11.5.6 Dealing with Oddly Named Files

Tcl commands use a leading dash to indicate an option to the command. For example, the `file copy` command accepts a `-force` option. You'll face problems, though, if the files you need to work with have names starting with dashes. That's because the Tcl commands assume the names starting

with dashes are options, not file names.
To handle cases like this, put a double dash, --, after the command and prior to the names of the files. For example, to copy a file named -force, use a command like the following:

```
file copy -force -- -force b.txt
```

This command forces the copy with the -force option, and then separates the command-line options from the file names with the --.

11.6 Reading and Writing Files

Tcl's history as a scripting language led to an emphasis on working with text files. Tcl was originally created on Unix systems where most information, including application data and system configuration settings, is stored in plain text files. Thus, most Tcl file commands assume that the content of the files you work on is text. However, Tcl provides the capability to read, write, and manipulate data in binary format as well.

11.6.1 Basic File I/O

The Tcl commands for file I/O are similar to the procedures in the C standard I/O library, both in their names and in their behavior. Here is a script called tgrep that illustrates most of the basic features of file I/O:

```
#!/usr/bin/env tclsh
if {$argc != 2} {
    error "Usage: tgrep pattern fileName"
}
set f [open [lindex $argv 1] r]
set pat [lindex $argv 0]
while {[gets $f line] >= 0} {
    if {[regexp -- $pat $line]} {
        puts $line
    }
}
close $f
```

This script behaves much like the Unix grep program, which searches files for text that matches a given pattern. You can invoke it from your shell with two arguments, a regular expression pattern and a file name, and it prints out

all of the lines in the file that match the pattern.
After making sure that it received enough arguments, the script invokes the `open` command on the file to search, which is the second argument to the script. `open` takes two arguments: the name of a file and an access mode. The access mode provides information such as whether you'll be reading the file or writing it, and whether you want to append it to the end of file or access it from the beginning. The access mode may have one of the following values:

- `r`—open for reading only. The file must already exist. This is the default if the access mode isn't specified.
- `r+`—open for reading and writing; the file must already exist.
- `w`—open for writing only. Truncates the file (deletes all existing content) if it already exists; otherwise creates a new empty file.
- `w+`—open for reading and writing. Truncates the file (deletes all existing content) if it already exists; otherwise creates a new empty file.
- `a`—open for writing only. Sets the initial access position to the end of the file. Thus, writing appends to the end of the existing content unless you reset the access position. If the file doesn't exist, creates a new empty file.
- `a+`—open the file for reading and writing. Sets the initial access position to the end of the file. If the file doesn't exist, creates a new empty file.

The access mode may also be specified as a list of POSIX flags like `RDONLY`, `CREAT`, and `TRUNC`. See the reference documentation for more information about these flags.
The `open` command returns a string such as `file3` that identifies the open file. This *file identifier* is used when invoking other commands to manipulate the open file, such as `gets`, `puts`, and `close`. Normally you save the file identifier in a variable when you open a file, and then use that variable to refer to the open file. You should not expect the identifiers returned by `open` to have any particular format.
Three file identifiers have well-defined names and are always available to you, even if you haven't explicitly opened any files. These are `stdin`, `stdout`, and `stderr`; they refer to the standard input, output, and error channels for the process in which the Tcl script is executing.

Note

On Windows, these default channels work only for console applications, not windowed applications.

The `file channels` command lists all open channels. For example, unless you have explicitly opened additional channels, this command typically returns only the standard I/O channels:

```
   file channels
⇒ stdin stdout stderr
```

After opening the file to search, the `tgrep` script reads the file one line at a time with the `gets` command. `gets` normally takes two arguments: a file identifier and the name of a variable. It reads the next line from the open file, discards the terminating end-of-line character or characters, stores the line in the named variable, and returns a count of the number of characters stored into the variable, not including the end-of-line sequence. If the end of the file is reached before any characters are read, `gets` stores an empty string in the variable and returns `-1`.

Note

Tcl also provides a second form of `gets` where the line is returned as the result of the command, and a `read` command for non-line-oriented input.

The `tgrep` script matches each line in the file against the pattern and prints it with `puts` if it matches. The `puts` command takes two arguments, which are a file identifier and a string to print. `puts` adds a newline character to the string and outputs the line on the given file. The script doesn't specify the output file identifier, so it uses the default `stdout` and the line is printed on standard output.
When `tgrep` reaches the end of the file, `gets` returns `-1`, which ends the `while` loop. The script then closes the file with the `close` command; this releases the resources associated with the open file. In most systems there is a limit on how many files may be open at one time in an application, so it is important to close files as soon as you are finished reading or writing to them. In this example the `close` is unnecessary, since the file is closed automatically when the application exits, but a good practice.

11.6.2 Output Buffering

The `puts` command uses the buffering scheme of the C standard I/O library. This means that information passed to `puts` may not appear immediately in the target file. In many cases (particularly if the file isn't a terminal device) output is buffered in the application's memory until a large amount of data has accumulated for the file, at which point all of the data will be written out in a single operation; 4KB buffers are common on many operating systems. (See Chapter 12 for more on buffering options.) If you need data to appear in a file immediately, you should invoke the `flush` command:

```
flush $f
```

The `flush` command takes a file identifier as its argument and forces any buffered output data for that file to be written to the file. `flush` doesn't return until the data has been written. Buffered data is also flushed when a file is closed.

11.6.3 Handling Platform End-of-Line Conventions

On Unix and Linux systems each line of text in a file ends with a newline character (ASCII decimal 10). Old Macintosh systems (predating Mac OS X) used a single carriage return character (ASCII decimal 13) instead; modern Macintosh systems follow Unix conventions. Windows systems, though, use two characters: a carriage return followed by a newline. Just accessing text files quickly becomes an exercise in operating system differences.

For the most part, Tcl handles these differences automatically. When writing to a file, the `puts` command automatically translates a newline character into the proper one- or two-character sequence, depending on the platform. Thus, you can always use `\n` in your Tcl scripts to indicate the end of a line.

Note

Remember that the `puts` command automatically appends an end-of-line sequence to the end of the string it writes. You can use the `-nonewline` option to suppress the end-of-line sequence.

By default, the `gets` command accepts any of the supported end-of-line conventions to determine the end of a line. The end-of-line convention can even change from line to line in the file. The `read` command functions the same way by default, and any end-of-line sequence read from the input is translated into a newline character in the string returned. Additionally, the `-nonewline` option tells the `read` command to discard the trailing newline in the string returned if the last character (or characters) consists of an end-of-line sequence.
You can use the `chan configure` command (or `fconfigure`, for compatibility with older versions of Tcl) to query or modify the default end-of-line behavior for any given channel or file; for example:

```
chan configure stdin -translation
⇒ auto
```

With no additional argument, this command returns the end-of-line translation mode for the channel. You can also provide an additional argument to set the translation mode of the channel. A single-element list sets the translation for both input and output, if the channel is bidirectional. You can also provide a two-element list where the first element specifies the input translation and the second element specifies the output translation.
By default, channels have a translation of `auto`. For input channels, `auto` means that all end-of-line sequences are treated as newlines; for output channels, the platform-specific convention is used automatically. A value of `cr`, `lf`, or `crlf` indicates to use only a newline, a carriage return, or a carriage return–newline sequence respectively for the end-of-line convention for the channel. For example, setting the translation to `crlf` on an output channel indicates that all newline characters should be translated to carriage return–newline sequences when written. In contrast, setting the translation to `cr` on an input channel indicates that only a carriage return character should be treated as an end of line, and when read it is translated into a newline character for use by Tcl. You can use a translation of `binary` to indicate that no end-of-line translations should take place; this also automatically sets the `-encoding` for the channel to `binary` (see Section 11.6.4).
Taking advantage of Tcl's automatic conversion of line endings, you can use the `read` command to create an alternate implementation of the previous `tgrep` script:

```
#!/usr/bin/env tclsh
if {$argc != 2} {?
    error "Usage: tgrep pattern fileName"
}
set f [open [lindex $argv 1] r]
set pat [lindex $argv 0]
set content [read $f]
close $f
foreach line [split $content "\n"]
    if [regexp -- $pat $line] {
        puts $line
    }
}
```

Note the use of the `split` command to split the string returned by `read` at each newline character. This results in a list of lines over which the `foreach` command can iterate. Through the use of `read`, this script can be much more efficient than the previous `tgrep` script, which used the `gets` command to access the file system for each line. With `read` the script loads the entire contents of the file into memory and then checks for the matching patterns. This is much more efficient for files that are not too large. But using this technique for very large files can use too much memory, which then can result in paging memory out to disk, significantly slowing your system.

11.6.4 Handling Character Set Encoding

As discussed in Chapter 5, Tcl represents all strings internally as Unicode characters in UTF-8 format. However, Tcl automatically converts strings from the internal UTF-8 format to the system encoding and vice versa whenever writing and reading text on a channel. So if you are working with a text file that uses the same encoding as the system encoding, you don't need to take any special steps for Tcl to handle the file correctly.
On the other hand, if the file doesn't use the same encoding as the system encoding (or you'd like to create a file with an encoding other than the system encoding), you can use the `-encoding` option to `chan configure` (or `fconfigure`, for compatibility with older versions of Tcl) to specify the encoding for Tcl to use for the file; for example:

```
chan configure $fid -encoding shiftjis
```

Note

To prevent Tcl from performing any character encoding translation when working with binary files, you should use `chan configure` to set the `-translation` to `binary`. Not only does this prevent platform-specific end-of-line character translation, but it also automatically sets the `-encoding` option on the channel to `binary` to prevent character encoding translation.

11.6.5 Working with Binary Files

Tcl assumes by default that all files are text files. As discussed in the previous section, if the file you are manipulating contains binary information, you should use `chan configure` to set the `-translation` to `binary`:

```
chan configure $fid -translation binary
```

To read from the file, you would then typically use the `read` command, specifying the number of bytes of data to read. You could then use the `binary scan` command, described in Chapter 5, to "unpack" the binary information into Tcl variables:

```
set data [read $fid 12]
binary scan $data iiss int1 int2 short1 short2
```

Conversely, to write binary data to a file, you would typically "pack" the values into a binary string with the `binary format` command, described in Chapter 5, and write the string to the file using the `puts` command. In most cases you should use the `-nonewline` option to `puts` to prevent it from automatically adding a newline character after the bytes written:

```
set data [binary format s3 {3 -7 1}]
puts -nonewline $fid $data
```

11.6.6 Random Access to Files

File I/O is sequential by default—each `gets` or `read` command returns the next

bytes after the previous `gets` or `read` command, and each `puts` command writes its data immediately following the data written by the previous `puts` command. However, you can use the `chan seek`, `chan tell`, and `chan eof` commands to access files nonsequentially. (Equivalent `seek`, `tell`, and `eof` commands are available for compatibility with older versions of Tcl.)
Each open file has an *access position*, which is the location in the file where the next read or write will occur. When a file is opened, the access position is set to the beginning or the end of the file, depending on the access mode you specified to `open`. After each read or write operation the access position increments by the number of bytes transferred. You can use the `chan seek` command to change the current access position. In its simplest form `chan seek` takes two arguments, a file identifier and an integer offset within the file. For example, the command

```
chan seek $f 2000
```

changes the access position for the file so that the next read or write will start at byte number 2000 in the file.

Note

The offset value is defined in terms of bytes, not characters. Thus, it may place the access position in the middle of a multibyte character sequence.

`chan seek` can also take a third argument that specifies an origin for the offset. The third argument must be either `start`, `current`, or `end`. `start` produces the same effect as if the argument were omitted: the offset is measured relative to the start of the file. `current` means that the offset is measured relative to the file's current access position, and `end` means that the offset is measured relative to the end of the file. For example, the following command sets the access position to 100 bytes before the end of the file:

```
chan seek $f -100 end
```

If the origin is `current` or `end`, the offset may be either positive or negative; for `start` the offset must be positive.

Note

It is possible to seek past the current end of the file, in which case the file may contain a hole. Check the documentation for your operating system for more information on what this means.

The `chan tell` command returns the current access position (in bytes, not characters) for a particular file identifier:

```
    chan tell $f
⇒ 186
```

This allows you to record a position and return to it later.
The `chan eof` command takes a file identifier as argument and returns `0` or `1` to indicate whether the most recent `gets` or `read` command for the file attempted to read past the end of the file:

```
    chan eof $f
⇒ 0
```

In most cases you'll use the `chan eof` command to detect the end of a file when using non-blocking channels. See Chapter 12 for more on this.

11.6.7 Copying File Content

Section 11.5.3 discussed the `file copy` command, which duplicates a file on disk. The `file copy` command is easy to use, but it does have its limitations: it creates a byte-for-byte duplicate of the original file and works only for on-disk files. A more flexible alternative is the `chan copy` command (or `fcopy`, for compatibility with older versions of Tcl), which allows transformations, such as changing the character encoding, and works with any supported Tcl channel type, such as pipes and sockets as well as on-disk files.
In its simplest form, the `chan copy` command accepts an input channel opened for reading and an output channel opened for writing as arguments. The input channel is read until an end of file is encountered, and the content is written to the output channel. Alternatively, you can use the `-size` option to specify a maximum number of bytes to read from the input channel. As an example, the following copies the contents of a file, translating from EUC-JP to Shift_JIS encoding:

```
set input [open $inFile r]
chan config $input -encoding euc-jp
set output [open $outFile w]
chan config $output -encoding shiftjis
chan copy $input $output
close $input
close $output
```

By default, the `chan copy` command blocks, not returning until the copy is complete; its return value is the number of bytes written to the output channel. You have the option of specifying a callback with the `-command` option, in which case the `chan copy` command returns immediately. The copy takes place in the background, and upon completion Tcl invokes the callback, appending one or two arguments. The first argument is the number of bytes written to the output channel. The second is present only if an error took place during the copy, in which case it consists of the error message.

Note

The background copy takes place only while Tcl's event loop is running. If the event loop is not already running, you'll need to start it with a command such as `vwait`. Additionally, you should not have any channel event handlers, as registered with the `chan event` or `fileevent` commands, active during the background copy. Chapter 12 has more information on the event loop, the `vwait` command, and channel event handlers.

The following code shows an example of a background copy:

```
proc copyComplete {args} {
    global enter_loop
    if {[llength $args] > 1} {
        puts "Error during file copy: [lindex $args 1]"
    } else {
        puts -nonewline "Copy completed successfully. "
    }
    puts "[lindex $args 0] bytes copied."
    set enter_loop 1
}
set input [open $inFile r]
chan config $input -encoding euc-jp
set output [open $outFile w]
chan config $output -encoding shiftjis
chan copy $input $output -command copyComplete
vwait enter_loop

# Copy operation has completed at this point

catch {close $input}
catch {close $output}
```

The `copyComplete` procedure implements the callback to execute once the file copy is complete. If only one argument is passed to it, the copy completed successfully; if more than one argument is provided, an error occurred during the copy, and the procedure prints the error message to the console. After the input and output channels are opened and configured, the `chan copy` command initiates the background copy. The `vwait` command starts the event loop; the `vwait` command waits until the global variable `enter_loop` is set (which is done during execution of the `copyComplete` callback script), at which point the event loop terminates and the `vwait` command returns. At this point, the two channels are closed.

11.7 Virtual File Systems

Starting with Tcl 8.4, Tcl's file commands support *virtual file systems*. This means that Tcl extensions can add support for other services to appear to Tcl as if they were file systems. For example, FTP network file access can appear to Tcl commands such as `glob` or `file` as if the remote files were really local on your hard disk, or files within a ZIP archive can be accessed as though they were individual files on the file system. Using such a virtual file system makes writing Tcl scripts a lot easier. You merely use the normal file commands in Tcl and let the Tcl extension perform all the difficult work. And, if the mapped file system supports it, you can create new files or edit existing ones on the mounted virtual file system, even if the underlying data is not a file. Some of the most exciting developments in Tcl, Starpacks and Starkits in particular, use virtual file systems. See Chapter 14 for more on these.

The `tclvfs` extension provides a set of bindings to virtual file systems that you can access from Tcl commands. These include ZIP archives, FTP network access, Metakit files (used by Starkits and Starpacks), as well as WebDAV and HTTP network protocols. One of the most common uses, though, is mounting compressed files as if they were file systems. This allows you to browse the contents of a ZIP archive, for example, or read individual files from it.

The following code mounts a ZIP file as if it were under a virtual directory named `zip` within the current working directory:

```
  puts [pwd]
⇒ /Users/ericfj/Documents/tcl
  package require vfs::zip
  vfs::zip::Mount tcl852-src.zip zip
```

This example uses the ZIP file of Tcl 8.5 sources downloaded from http://tcl.sourceforge.net; you can use any ZIP files you have handy instead. Once it is mounted, you can access a virtual file system as if it were a normal file system. For example, to list the top-level files in the Tcl source distribution just mounted, you could use a command like the following:

```
foreach f [glob zip/tcl8.5.2/*] {
    puts $f
}
```

In this example, the `glob` command returns all files underneath the directory `zip/tcl8.5.2`. The `zip` part comes from where we mounted the virtual file system. The `tcl8.5.2` part comes from the top-level directory inside the ZIP file.

Note

If the `package require` command fails, you need to install the `tclvfs` extension. The easiest way to do this is to download the ActiveTcl release, which includes a precompiled version of `tclvfs`. You can also download the source code for `tclvfs` from http://sourceforge.net/projects/tclvfs.

Once you have mounted virtual file systems, you can use the `vfs::filesystem info` command to get a list of all mounted ones; for example:

```
    puts [vfs::filesystem info]
⇒ {/Users/ericfj/Documents/tcl/zip}
```

When you're finished with a virtual file system, use the `vfs::filesystem unmount` command to unmount it:

```
vfs::filesystem unmount zip
```

As another example, consider mounting an `mk4`, or Metakit, file as a virtual file system. Tcl Starkits and Starpacks are Metakit files. You can download a Starkit file, such as `tclhttpd.kit` (an HTTP server written in Tcl), from sites such as http://tcl.tk/starkits. This file provides a good example of a Metakit file, along with a nice Tcl source code example that shows how to build an HTTP (web) server.
You can access the contents of a Starkit file similarly to a ZIP file; for example:

```
package require vfs::mk4
vfs::mk4::Mount tclhttpd.kit kit
foreach f [glob kit/*] {
    puts $f
}
vfs::filesystem unmount kit
```

When you run this script, you'll see output such as the following:

```
⇒ kit/bin
⇒ kit/custom
⇒ kit/htdocs
⇒ kit/lib
⇒ kit/main.tcl
```

See the reference documentation on `vfs` and `vfs-filesystems` for more information on how to use virtual file systems.

11.8 Errors in System Calls

Most of the commands described in this chapter invoke calls on the operating system, and in many cases the system calls can return errors. This can happen, for example, if you invoke `open` or `file stat` on a file that doesn't exist, or if an I/O error occurs in reading a file. The Tcl commands detect these system call errors and in most cases return errors themselves. The error message identifies the error that occurred:

```
    open bogus
  ∅ couldn't open "bogus": no such file or directory
```

When an error occurs in a system call, Tcl also sets the `errorCode` global variable and the `-errorCode` key in the return options dictionary to provide additional information about the error. The value is typically a three-element list, where the first element is the string `POSIX`, the second element is a symbolic error name such as `ENOENT`, and the last element is a human-readable error message. See Chapter 13 for more information on error handling.

12. Processes and Interprocess Communication

Tcl provides several commands for dealing with processes. You can create new processes with the `exec` command, or you can create new processes with `open` and then use file I/O commands to communicate with them. Tcl also supports interprocess communication via TCP/IP sockets. You can access process identifiers with the `pid` command. You can read and write environment variables using the `env` variable, and you can terminate the current process with the `exit` command.

12.1 Commands Presented in This Chapter

This chapter discusses the following commands related to process control and interprocess communication:

- `exec ?-keepnewline? ?-ignorestderr? ?--?` *`arg`* `?`*`arg`* `...?`

Executes the command pipeline specified by *`arg`*s using one or more subprocesses and returns the pipeline's standard output or an empty string if output is redirected (the trailing newline, if any, is dropped unless `-keepnewline` is specified). I/O redirection and pipes may be specified. If the last *`arg`* is `&`, the pipeline is executed in background and the return value is a list of its process IDs. If any process writes to its standard error channel and that output is not explicitly redirected, `exec` raises an error unless the `-ignorestderr` option is specified.

- `exit ?`*`code`*`?`

Terminates a process, returning *`code`* to the parent as the exit status. *`code`* must be an integer and defaults to `0`.

- `open |`*`command`* `?`*`access`*`?`

Treats *`command`* as a list with the same structure as arguments to `exec` and creates subprocess(es) to execute command(s). Depending on *`access`*, it creates pipes for writing input to a pipeline and/or reading output from it. Returns a file identifier for communicating with the subprocesses.

- `pid ?`*`fileID`*`?`

If *`fileID`* is omitted, returns the process identifier for the current process. Otherwise it returns a list of all the process IDs in the pipeline associated with `fileID` (which must have been opened using |).

- `socket ?options? host port`

Opens a client-side TCP/IP socket connection to the `host` and `port` indicated, returning a channel identifier for the socket connection created. The `host` may be either a domain-style host name or a numerical IP address. The `port` may be an integer port number (or a service name, where supported and understood by the host operating system). See the following sections and the reference documentation for a description of the supported options.

- `socket -server command ?options? port`

Opens a TCP/IP server socket for the `port`, returning the channel identifier of the server socket created. The `port` may be an integer port number (or a service name, where supported and understood by the host operating system); a value of `0` causes the operating system to allocate an unused port for the server socket. When a client connects to the server port, Tcl automatically creates a new channel to use for communication with that client, then invokes the specified `command` with three additional arguments: the new client communication channel, the client IP address, and the client port number. Closing the server channel shuts down the server so that no new connections are accepted. See the following sections and the reference documentation for a description of the supported options.

-
  ```
  chan configure channelID ?optionName? ?value?
                 ?optionName value ...?
  fconfigure channelID ?optionName? ?value?
                 ?optionName value ...?
  ```

Queries or sets the configuration options of the channel `channelID`. If no `optionName` or `value` arguments are supplied, the command returns a dictionary of all configuration options and their values. If `optionName` is supplied but no `value`, the command returns the current `value` of the given option. If one or more pairs of `optionName` and `value` are supplied, the command sets each of the named options to the corresponding value; in this case the return value is an empty string. See the following sections and the reference documentation for a description of the supported options. Starting with Tcl 8.5, `fconfigure` is deprecated in favor of `chan configure`.

-
  ```
  chan event channelID event ?script?
  fileevent channelID event ?script?
  ```

Installs the Tcl script `script` as a file event handler to be called whenever `channelID` enters the state described by `event` (which must be either `readable` or `writable`). Only one such handler may be installed per event per channel at a time. If `script` is an empty string, the current handler is deleted. If `script` is omitted, the currently installed script is returned (or an empty string if no such handler is installed). Starting with Tcl 8.5, `fileevent` is deprecated in favor of `chan event`.

- `vwait` *`variableName`*

Enters the Tcl event loop, and continues to process all events received until some event handler sets the value of the variable specified by *`variableName`*. At that point, once the event handler that set the variable completes, the `vwait` command finally returns.

12.2 Terminating the Tcl Process with `exit`

Invoking the `exit` command terminates the process in which the command was executed. `exit` takes an optional integer argument. If this argument is provided, it is used as the exit status to return to the parent process. `0` indicates a normal exit, and nonzero values correspond to abnormal exits; values other than `0` and `1` are rare. If no argument is given to `exit`, it exits with a status of `0`. Since `exit` terminates the process, it doesn't have any return value.

12.3 Invoking Subprocesses with `exec`

The `exec` command creates one or more subprocesses and waits until they complete before returning. For example,

```
exec rm main.o
```

executes `rm` as a subprocess, passes it the argument `main.o`, and returns after `rm` completes. The arguments to `exec` are similar to what you would type as a command line to a shell program such as `sh` or `csh`. The first argument to `exec` is the name of a program to execute, and each additional argument forms one argument to that subprocess.

Note

The preceding example is for illustration only. If your goal is to delete a file, copy a file, list the contents of a directory, etc., Tcl's built-in commands such as `file delete` are a better choice for two reasons. First, by avoiding platform-specific commands, such as `rm` on Unix and `del` on Windows, you can run your script on any operating system where Tcl is supported. Second, the built-in Tcl command deletes the file by invoking a system function call, rather than incurring the expense of starting another process.

To execute a subprocess, `exec` looks for an executable file with a name equal to `exec`'s first argument. If the name contains a `/` or starts with `~`, `exec` checks the single file indicated by the name. Otherwise `exec` checks each of the directories in the `PATH` environment variable to see if the command name refers to an executable file in that directory. `exec` uses the first executable that it finds.

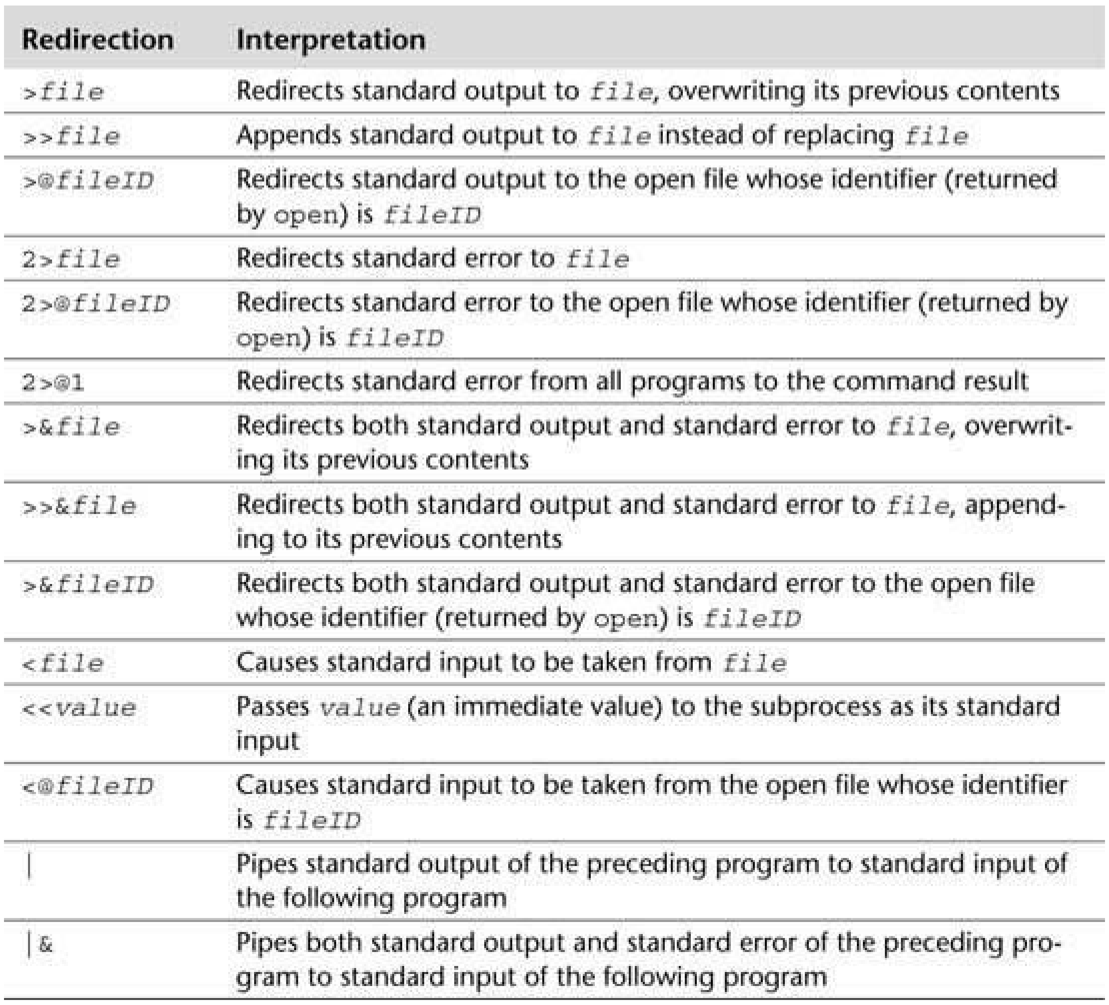
`exec` collects all of the information written to standard output by the subprocess and returns that information as its result, as in the following example:

```
    exec wc /usr/include/stdio.h
⇒        71      230     1732 /usr/include/stdio.h
```

If the last character of output is a newline, `exec` removes it. This behavior may seem strange, but it makes `exec` consistent with other Tcl commands, which don't normally terminate the last line of the result; you can retain the newline by specifying `-keepnewline` as the first argument to `exec`.

`exec` supports I/O redirection in a fashion similar to the Unix shells. For example, if one of the arguments to `exec` is `>foo`, output from the process is placed in the file `foo` instead of returning to Tcl as `exec`'s result. In this case `exec`'s result will be an empty string. Many other forms of input and output redirection are supported as well, as listed in Table 12.1.

Table 12.1 Subprocess I/O Redirection Syntax



Redirection	Interpretation
>*file*	Redirects standard output to *file*, overwriting its previous contents
>>*file*	Appends standard output to *file* instead of replacing *file*
>@*fileID*	Redirects standard output to the open file whose identifier (returned by open) is *fileID*
2>*file*	Redirects standard error to *file*
2>@*fileID*	Redirects standard error to the open file whose identifier (returned by open) is *fileID*
2>@1	Redirects standard error from all programs to the command result
>&*file*	Redirects both standard output and standard error to *file*, overwriting its previous contents
>>&*file*	Redirects both standard output and standard error to *file*, appending to its previous contents
>&*fileID*	Redirects both standard output and standard error to the open file whose identifier (returned by open) is *fileID*
<*file*	Causes standard input to be taken from *file*
<<*value*	Passes *value* (an immediate value) to the subprocess as its standard input
<@*fileID*	Causes standard input to be taken from the open file whose identifier is *fileID*
|	Pipes standard output of the preceding program to standard input of the following program
|&	Pipes both standard output and standard error of the preceding program to standard input of the following program

Note

The `@fileID` syntax is not supported on Windows. Additionally, not all Windows applications work well with the `exec` command.

In each of the cases shown in the table, `file` or `value` or `fileID` may follow the redirection symbol in a single argument, or it may be in a separate argument. As an example of using redirection, consider the following command:

```
exec cat << "test data" > foo
```

This command overwrites the file `foo` with the string `test data`. The string is passed to `cat` as its standard input; `cat` copies the string to its standard output, which has been redirected to file `foo`. If no input redirection is specified, the subprocess inherits the standard input channel from the Tcl application; if no output redirection is specified, the standard output from the subprocess is

returned as `exec`'s result.
You can also invoke a pipeline of processes instead of a single process, as in the following example:

```
    exec grep #include tclInt.h | wc
⇒          8         25       212
```

The `grep` program extracts all the lines containing the string `#include` from the file `tclInt.h`. These lines are then piped to the `wc` program, which computes the number of lines, words, and characters in the `grep` output and prints this information on its standard output. The `wc` output is returned as the result of `exec`.
If the last argument to `exec` is `&`, the subprocess(es) will be executed in background. `exec` returns immediately, without waiting for the subprocesses to complete. It returns a list containing the process identifiers for all of the processes in the pipeline; standard output and standard error from the subprocesses go to the standard output and standard error of the Tcl application unless redirected.
The `exec` command raises an error if a subprocess is suspended or exits abnormally (for example, it is killed or returns a nonzero exit status). `exec` also raises an error if a subprocess generates output on its standard error channel and standard error was not redirected; you can suppress this condition by providing the `-ignorestderr` option to `exec`. The error message consists of the output generated by the last subprocess (unless it was redirected with `>`), followed by an error message for each process that exited abnormally, followed by the information generated on standard error by the processes, if any. In addition, `exec` sets the `errorCode` variable and the `-errorcode` key in the return options dictionary to hold information about the last process that terminated abnormally, if any (see Section 13.2 and the reference documentation for details).

Note

Many Unix programs are careless about the exit status that they return. If you invoke such a program with `exec` and it accidentally returns a nonzero status, the `exec` command generates a false error. To prevent these errors from aborting your scripts, invoke `exec` inside a `catch` command as described in Chapter 13.

Although `exec`'s features are similar to those of the Unix shells, there is one important difference: `exec` does not perform any file name expansion. For example, suppose you invoke the following command with the goal of removing all `.o` files in the current directory:

```
   exec rm *.o
Ø rm: *.o nonexistent
```

`rm` receives `*.o` as its argument and exits with an error when it cannot find a file by this name. If you want file name expansion to occur, you can use the `glob` command to get it, but not in the obvious way. For example, the following command will not work:

```
   exec rm [glob *.o]
Ø rm: a.o b.o nonexistent
```

This fails because the list of file names that `glob` returns is passed to `rm` as a single argument. If, for example, there exist two `.o` files, `a.o` and `b.o`, `rm`'s argument will be `a.o b.o`; since there is no file by that name, `rm` returns an error. The recommended way to handle this situation is to use Tcl's argument expansion syntax to provide the list elements as separate arguments:

```
exec rm {*}[glob *.tmp]
```

As discussed in Section 2.8, versions of Tcl prior to 8.5 used the `eval` command to similar effect:

```
eval exec rm [glob *.o]
```

12.4 I/O to and from a Command Pipeline

You can also create subprocesses using the `open` command; once you've done this, you can use commands like `gets` and `puts` to interact with the pipeline. Here are two simple examples:

```
set f1 [open {|sort -k 2 > newfile.txt} w]
set f2 [open |prog r+]
```

If the first character of the "file name" passed to `open` is the pipe symbol `|`, the argument isn't really a file name at all. Instead, it specifies a command pipeline. The remainder of the argument after the `|` is treated as a list whose elements have exactly the same meaning as the arguments to the `exec` command. `open` creates a pipeline of subprocesses just as for `exec`, and it returns an identifier that you can use to transfer data to and from the pipeline. In the first example the pipeline is opened for writing, so a pipe is used for standard input to the Unix `sort` program, and you can invoke `puts` to write data on that pipe; the output from `sort` is redirected to a file named `newfile.txt`. The second example opens a pipeline for both reading and writing, so separate pipes are created for `prog`'s standard input and standard output. Commands like `puts` can be used to write data to `prog`, and commands like `gets` and `read` can be used to read the output from `prog`. In general, any of the I/O-related commands discussed in Chapter 11 can be used to interact with a pipeline channel.

Note

When writing data to a pipeline, don't forget that output is buffered. It probably will not be sent to the child process until you invoke the `chan flush` command to force the buffered data to be written. You can change the channel's buffering using the `chan configure` command, as discussed in Section 12.5.2.

When you close a file identifier that corresponds to a command pipeline, the `close` command flushes any buffered output to the pipeline, closes the pipes leading to and from the pipeline, if any, and waits for all of the processes in the pipeline to exit. If any of the processes exits abnormally, `close` returns an error in the same way as `exec`.

12.5 Configuring Channel Options

As discussed in Chapter 11, the `chan configure` command sets channel options (as does the `fconfigure` command, for compatibility with older versions of Tcl). Chapter 11 discussed the use of this command to configure character encoding and end-of-line translations for file I/O. You can use `chan configure` to configure these settings for command pipelines and socket channels as well. This section describes some additional `chan configure` settings of particular interest to command pipelines and socket channels.

12.5.1 Channel Blocking Mode

By default, when you call `gets` or `read` on a channel, Tcl blocks until data is available to read. `gets` blocks until it has received a complete line of data, and `read` blocks until it has read the specified number of characters or it encounters the end of file. Similarly, the `puts` command buffers the output, and if the buffers fill, `puts` blocks until the output gets sent. This is called *blocking* mode. Blocking forms the proper option for many uses and simplifies programming. For example, your program may read in the contents of a file, process the data, and output the results. Blocking works fine in most cases for this usage.
However, if your application opens a process pipeline or network socket, data might arrive at sporadic intervals, and blocking I/O commands could pause the process for an indeterminate amount of time. If your application has a graphical user interface, Tcl can't service key presses, button clicks, window refreshes, and other interactions while an I/O command is blocked; your application is frozen until the I/O command finally returns. In these types of applications, using non-blocking channels is usually preferable. Similarly, network servers usually need to run in non-blocking mode to be able to handle multiple clients at once. A server cannot wait until one client sends data before servicing the remaining clients, especially if the client the server waits for is not currently sending data; other clients are then starved for service in blocking mode.
The `-blocking` option of `chan configure` indicates whether or not the channel is configured for blocking mode. You can change the option by providing a Boolean value. For example, to set a channel to non-blocking mode:

```
chan configure $chan -blocking 0
```

If you do not pass a value to `chan configure`, the command returns the current value for an option. For example, to determine if a channel is set to blocking mode, issue the following command:

```
chan configure $socketid -blocking
```

When a channel is set to non-blocking mode, a `read` on the channel returns all characters available on the channel, up to the maximum number of characters requested, if specified. A `gets` on a non-blocking channel returns characters only if there is a complete line to read. If there is not a complete line, `gets` does not consume any characters. If you didn't provide a variable

name as an argument, `gets` returns an empty string; if you did provide a variable name, `gets` sets the variable to an empty string and returns `-1`. This is actually the same result that `gets` provides in the case of an end-of-file condition, so Tcl provides two additional commands to distinguish between these cases.
The `chan eof` command returns `1` if the last input operation on a channel encounters an end-of-file condition, and `0` otherwise. The `chan blocked` command returns `1` if the most recent input operation on a channel returns less information than requested because all available input is exhausted, and `0` otherwise. (The `eof` and `fblocked` commands are also available for compatibility with older versions of Tcl.)

12.5.2 Channel Buffering Mode

Tcl automatically buffers all channel I/O for efficiency. The channel's input and output buffers have the same size, typically defaulting to 4KB. You can view or change the buffer size for a channel using the `-buffersize` option of `chan configure`. You can change the buffer size to any value from 1 byte to 1MB currently; values requested outside of this range are appropriately adjusted to the minimum or maximum supported.
The buffer size is the only configurable parameter for input buffering. If there is not sufficient data in an input buffer to fulfill a read request on a channel, Tcl reads as much data as is currently available from the underlying source (e.g., file, pipeline, socket, etc.) up to the maximum buffer size.
The behavior of output buffering on a channel is also controlled by the `-buffering` option to `chan configure`, which determines when Tcl should automatically flush a channel. The `-buffering` option supports values of `none` (flush on every output), `line` (flush after every line), or `full` (flush when the buffer fills or the script calls the `chan flush` command). The `-buffering` option defaults to `full` except for channels that connects to "terminal-like" devices; for these channels the initial setting is `line`. Additionally, `stdin` and `stdout` are initially set to `line`, and `stderr` is set to `none`.

12.6 Event-Driven Channel Interaction

The default blocking behavior of channels described in Section 12.5.1 is usually not a problem for many applications. If the application is driven from the console rather than a graphical user interface, or if the application doesn't interact with the user at all, it usually doesn't matter if an I/O operation on the channel is blocked for seconds or even minutes at a time. On the other hand, if the application must present a responsive user interface, must interact with several channels simultaneously, or needs to perform other operations in parallel with the channel access, having the entire application freeze while an I/O operation blocks is unacceptable. Changing the channels to non-blocking helps to address this issue. But even then, attempts to read from the channels might return with no data if there is no data available from the underlying source. Repeatedly attempting to read from the channel until data is available would waste processing time.

Many other languages use threads to address this issue of interacting with blocking channels. If one thread is blocking by an I/O operation, other threads are free to continue execution. Multithreaded programs have their own issues, though, such as managing resource contention and synchronization. Although Tcl does have a `Thread` extension, which exposes multithreaded programming at the scripting level (see Appendix B), traditionally Tcl has taken a different approach: *event-driven* programming.

In an event-driven programming model, an application registers scripts as *event handlers* that are triggered in response to various *events* that take place during its execution. Chapter 1 has already introduced the concept of event handlers to implement widget action in response to user interaction. Using these event *bindings* in Tk to associate actions with user interaction and other windowing events is further discussed in Chapter 17 and Chapter 22. Tcl also supports timer-related events through a command called `after`, which is discussed in Chapter 15.

Tcl supports another set of bindings to respond to events that take place on channels, which are referred to as *file events* or *channel events*. For example, you can register a script to be invoked whenever data arrives on a pipeline or socket channel. If your application needs to interact with multiple channels, each can have its own file event handlers to process those events in whatever way is appropriate. At the same time, Tcl can also monitor windowing and timer events and invoke any event handlers associated with those events as well.

12.6.1 Entering the Tcl Event Loop with `vwait`

To be able to detect events occurring and to invoke any registered handlers, Tcl must be in its *event loop*. When an application is invoked with the `wish` interpreter or has the Tk package loaded, it automatically enters the event

loop after executing all of the commands in the script file. For applications that don't use Tk, you must explicitly enter the Tcl event loop with a command.
Typically the `vwait` command is used to invoke the event loop. `vwait` accepts one argument, the name of a global or persistent namespace variable, as in

```
vwait enter_loop
```

The `vwait` command enters the Tcl event loop and continues to process all events received until some event handler sets the value of the variable specified (`enter_loop` in this example). At that point, once the event handler that set the variable completes, the `vwait` command finally returns.

12.6.2 Registering File Event Handlers

You register a file event handler with the `chan event` command. (The `fileevent` command is also available for compatibility with older versions of Tcl.) In addition to specifying the channel identifier and a script to invoke when the event occurs, you indicate whether you are registering a handler for a readable or writable event on the channel. For example, the following arranges to call the command `ReadLine` whenever there is data to read on the channel whose identifier is stored in the variable `pipe`:

```
chan event $pipe readable ReadLine
```

You can create at most one readable and one writable event handler for a given channel. Invoking `chan event` to install an event handler of a given type on a channel replaces any existing handler of that same type on the channel. If you omit the script argument, `chan event` returns the currently installed handler, if any.
The Tcl event loop must be active for Tcl to be able to detect and handle file events. Additionally, you normally should use the `chan configure` command to place the channel in non-blocking mode. Otherwise, the handler could block when attempting to read from or write to the channel.
A channel is considered to be writable if at least 1 byte of data can be written to the underlying file or device without blocking, or if an error condition is present on the underlying file or device. Writable file event handlers are used most frequently to detect when a client socket opened in asynchronous mode becomes connected or if the connection fails. This is discussed in Section 12.9.

A channel is considered to be readable if there is unread data available on the underlying device. A channel is also considered to be readable if there is unread data in an input buffer, except in the special case where the most recent attempt to read from the channel was a `gets` call that could not find a complete line in the input buffer. This feature allows a file to be read a line at a time in non-blocking mode using events. A channel is also considered to be readable if an end-of-file or error condition is present on the underlying file or device.

Note

It is important for an event handler to check for these conditions and handle them appropriately. For example, if there is no special check for end of file, an infinite loop may occur when the handler reads no data, returns, and is invoked again immediately.

The following script is a simple example of using file events to process the output of a process pipeline. It illustrates several features of handling asynchronous communication on a channel in a safe and robust manner.

```
set fid [open "| du /usr " r]
chan configure $fid -blocking 0
chan event $fid readable [list ReadLine $fid]

proc ReadLine {fid} {
    global done
    if {[catch {gets $fid line} len] || [chan eof $fid]} {

        # The last gets encountered EOF or we had an
        # abnormal termination.

        catch {close $fid}
        puts stderr "Channel closed"
        set done 1
        return
    } elseif {$len >= 0} {

        # We read a complete line, otherwise gets would have
        # read no characters and returned -1.

        puts "Read line: $line"
    }

    # We reach this point if there wasn't a complete line
    # to read. Simply return to the event loop to wait
    # for more data.
}

vwait done
```

The first line of the script opens a process pipeline for read access. In this case, it starts the Unix `du` command, which recursively reports disk usage for a directory. The script then configures the channel for non-blocking access. If this were a bidirectional channel where you were writing to the channel to send data to the process pipeline, you might also want to configure the buffering on the channel, perhaps to `line` mode so that each line would be written to the pipe immediately.
The script then registers a file event handler to invoke whenever data is available to read on the channel. In this case, the script is a well-formed list consisting of the `ReadLine` command and the channel identifier as an argument. For a simple handler such as this, we could have just quoted the handler script in double quotes to allow substitution of the variable, because channel identifiers don’t include whitespace or other special characters. However, if we had tried to pass other variable values as arguments that could be arbitrary strings, simple double quoting could produce malformed commands. When registering callbacks, it’s safest to use techniques such as the `list` command to ensure that the resulting script argument is a well-formed Tcl command. After defining the event handler procedure, the script

finally executes `vwait` to enter the event loop. Tcl continues to process events in the event loop until a handler sets the variable `done` to any value, at which point the `vwait` command returns and the script terminates.
The `ReadLine` handler itself first tries to read a complete line of data from the channel. The `gets` command could fail if the channel shuts down abnormally, so it's safest to execute it in a `catch` statement. If `catch` reports an error, or if an end-of-file condition is detected on the channel, we've read all data possible from the channel. The handler closes the channel (using a `catch` to silently discard any errors), sets the global variable `done` to `1`, and returns to the event loop. Because the `vwait` command is waiting for `done` to be set, this terminates the event loop.
On the other hand, if an error or end of file is not detected, `Readline` checks the return value of the `gets` command. If it was `-1`, `gets` was not able to read a complete line, and so it consumed no characters from the channel. (The other situation in which `gets` would return `-1`, an end of file, was tested for previously.) In this case, `ReadLine` simply returns to the event loop to wait for more data to arrive. But if `gets` was able to read a complete line, the characters read are now contained in the variable `line` and can be processed as desired.

12.7 Process IDs

Tcl provides three ways to access process identifiers. First, invoking a pipeline in background using `exec` returns a list containing the process identifiers for all of the subprocesses in the pipeline. You can use these identifiers, for example, if you wish to kill the processes. Second, you can invoke the `pid` command with no arguments and it returns the process identifier for the current process. Third, you can invoke `pid` with a file identifier as an argument, as in the following example:

```
    set f [open {| tbl | ditroff -ms} w]
    pid $f
 ⇒ 7189 7190
```

If there is a pipeline corresponding to the open file, as in the example, the `pid` command returns a list of identifiers for the processes in the pipeline.

12.8 Environment Variables

As mentioned in Chapter 3, the global array variable `env` contains all of the environment variables as elements, where the name of the element in `env` corresponds to the name of the environment variable. If you modify the `env` array, the changes are reflected in the process's environment variables and the new values are also passed to any child process created with `exec` or `open`.

12.9 TCP/IP Socket Communication

Sockets link applications using network protocols such as TCP/IP, the Internet's transmission control protocol/Internet protocol. Because socket-based networking is standardized, Tcl's support for sockets enables another avenue for integrating your Tcl applications with other, non-Tcl, applications.
Sockets are identified by a host name or IP address and a port number. A *port* is sort of like a channel number on a television. The port numbers for most network services, such as HTTP, are standardized (port 80 for HTTP, for example). Normally, port numbers under 1024 are restricted for use by privileged applications. In addition, many port numbers between 1024 and 49151 inclusive are standardized, such as 6000 and 6001 for the X Window System, and are often registered with the Internet Assigned Numbers Authority (IANA). Port numbers higher than this are not standardized but may be in use by other applications installed on your system. On Unix systems, look at the file `/etc/services` for a listing of predefined port numbers.
Unlike most systems, Tcl's communications channels are driven by events. Typical servers written in other languages either fork (clone) the current process to handle an incoming connection or spawn a new thread. Tcl servers, on the other hand, typically use event handler procedures to process network requests. See Section 12.6 for more on this subject.

12.9.1 Creating Client Communication Sockets

To create a socket, use the `socket` command. You can use this command to create a socket either for a client application to connect to a server, or for a server application that awaits client requests. To create a client socket you need to provide the host name or IP address and port number as arguments:

```
set sockid [socket localhost 12345]
```

This command opens a client-side socket on the local system on port 12345. The returned channel identifier gets stored in `sockid`.

Note

Use `localhost` to refer to the computer where your Tcl program executes.

With client sockets, you can use the `-myaddr` option to specify the Internet address of a network interface on the local system to use, which is handy for systems with more than one network interface (or network adapter card). Use `-myport` to specify the client-side port, if needed; if this option is omitted, the client's port number is chosen at random by the system software, which is typical for client-side sockets.

Once you have established the socket connection, you can then configure the channel and interact with it using I/O commands just as with process pipelines. For example, you could configure it to non-blocking mode and register a file event handler to read data from the socket as it arrives, just as in the process pipeline example shown in Section 12.6.2. By default, socket channels are configured for blocking and full buffering. The `-translation` defaults to `{auto crlf}` and the `-encoding` defaults to the system encoding. See Section 11.6 for more information on these settings and using the `chan configure` command to change them.

For a socket channel, the `chan configure` command supports two additional read-only options to report information about the socket connection. The value of the `-sockname` is a three-element list consisting of the IP address, the host name, and the port number for the socket. The value of the `-peername` option contains a three-element list with the same information for the peer socket.

By default, the `socket` command doesn't return until the socket connection is established. You can also specify `-async` to create a connection asynchronously. This means the `socket` command returns before the connection has been established. If the socket is configured for blocking mode, the next `gets`, `read`, or `chan flush` command waits until the socket is connected. If the socket is not in blocking mode, the next `gets`, `read`, or `chan flush` returns immediately, and a subsequent call to `chan blocked` would return `1`. Any error establishing the socket connection is reflected as an error when these commands are invoked on the channel. You can use the `chan event` command to register a `writable` event handler to detect when the connection has been established; client sockets opened in asynchronous mode become `writable` when they become connected or if the connection fails. You can then test for the presence of the `-peername` option in the channel's configuration to determine if the connection was successful, as shown here:

```
proc connected {fid} {

    # Remove the writable event handler to prevent it
    # from being invoked again.

    chan event $fid writable {}

    # Test for the existence of -peername in the channel
    # configuration to see if the connection was successful.

    if {[dict exists [chan configure $fid] -peername]} {
        # We're connected. Do whatever you like.
    } else {
        # We're not connected. Clean up and report the
        # situation however you like.
    }
}

set fid [socket -async $host $port]
chan event $fid writable [list connected $fid]
```

12.9.2 Creating Server Sockets

To create a server socket, call `socket` with the `-server` option:

socket -server *command* ?*options*? *port*

This command establishes a server socket, also known as a *listening socket*, on the current host at the given port number. You can also pass a service name in place of a port number, whose value is dependent on your operating system. For systems with multiple network interfaces, you might also find the `-myaddr` option useful to specify the host name or IP address of a specific network interface to use; otherwise the server socket is bound so that it can accept connections from any interface.
When the server socket detects a client connection, Tcl creates a channel for reading and writing to the client and calls the *`command`* you provided with three additional arguments: the new socket identifier, the client's address, and the client's port number. (Tcl's event loop must be running for the server

socket to detect connection requests.) Once the client is connected, you can use normal channel commands, such as those described in Section 12.9.1, to configure the channel and communicate with the client. Because each client connection is assigned a unique socket channel identifier, the server can manage multiple client connections simultaneously. You can use arrays, dictionaries, or other data structures to store information related to each client; the socket's channel identifier makes an excellent unique key in this case.

The following script shows a complete implementation of a simple multiclient echo server:

```
set listen [socket -server ClientConnect 9001]

proc ClientConnect {sock host port} {
    chan configure $sock -buffering line -blocking 0
    chan event $sock readable [list ReadLine $sock]
    SendMessage $sock "Connected to Echo server"
}

proc ReadLine {sock} {
    if {[catch {gets $sock line} len] || [eof $sock]} {
        catch {close $sock}
    } elseif {$len >= 0} {
        EchoLine $sock $line
    }
}

proc EchoLine {sock line} {
    global forever
    switch -nocase -- $line {
        exit {
            SendMessage $sock "Killing Echo server"
            catch {close $sock}
            set forever 1
            }
            quit {
                SendMessage $sock \
                    "Closing connection to Echo server"
                catch {close $sock}
            }
            default {
                SendMessage $sock $line
            }
        }
    }

    proc SendMessage {sock msg} {
        if {[catch {puts $sock $msg} error]} {
            puts stderr "Error writing to socket: $error"
            catch {close $sock}
        }
    }

    vwait forever
    catch {close $listen}
```

The script creates a listening socket on port 9001 of the local system. Whenever a client connection request is detected, Tcl calls the `ClientConnect` procedure, which configures the communication channel with the client, registers a readable file event handler for the channel, and then sends a connection confirmation message to the client. Note that the socket's channel identifier is included as an argument to the `ReadLine` procedure registered as the event handler. As a result, the `ReadLine` procedure knows which channel has data available to read when it is invoked, and we can use the same procedure for all channels.

The `ReadLine` procedure is very similar to the event handler presented in Section 12.6.2 for process pipeline channels. It closes the communication socket if the client disconnects or an error occurs while reading from the channel. Otherwise, it checks whether it read a complete line from the channel, and if so it passes the line read off to the `EchoLine` procedure for processing; otherwise, `ReadLine` returns to the event loop to wait for more data to arrive on the channel.

The `EchoLine` procedure checks to see if the line just read is a special control message. A line consisting of just the string `exit` terminates the echo server by setting the `vwait` variable and returning. A line consisting of `quit` causes the echo server to send an acknowledgment to the client and closes the client's socket channel, disconnecting the client. Any other line received is simply echoed back to the client.

The `SendMessage` procedure is a helper procedure to send messages to a client safely. If an error is detected sending the message, the socket is dead. The procedure logs a message to the standard error channel, then closes the socket channel.

Note

The extensive use of `catch` commands in this example is a good practice when writing multiclient servers. You don't want a communication glitch with one client to bring down the entire server. Use the `catch` command to guard against errors when opening channels and in all I/O operations with the channels.

12.10 Sending Commands to Tcl Programs

TCP/IP sockets let you send data between applications, and the Tk package also allows you to send commands directly from one Tk-based application to another on most Unix systems. With the `send` command, any Tk application can invoke arbitrary Tcl scripts in any other Tk application on the display; these commands can both retrieve information and take actions that modify the state of the target application.

Note

The `send` command is part of Tk, not the base Tcl language.

The `send` command is built on top of data-transfer facilities in the X Window System, used on most Unix systems for graphics. `send` is not available on systems that don't use the X Window System.

Note

In most cases, you should use sockets to pass data between applications. Not only are sockets safer, but Tcl's sockets work on multiple operating systems.

12.10.1 Basics of `send`

To use `send`, all you have to do is give the name of an application and a Tcl script to execute in the application. For example, consider the following command:

```
send myapp {selectLine 200}
```

The first argument to `send` is the name of the target application, and the second argument is a Tcl script to execute in that application. Tk locates the named application, forwards the script to that application, and arranges for the script to be executed in the application's interpreter. The result or error generated by the script is passed back to the originating application and returned by the `send` command.
`send` is synchronous: it doesn't complete until the script has been executed in the remote application and the result has been returned. `send` defers the processing of X events while it waits for the remote application to respond, so the application does not respond to its user interface during this time. After the `send` command completes and the application returns to normal event processing, any waiting events are processed. A sending application *will* respond to `send` requests from other applications while waiting for its own `send` to complete. This means, for example, that the target of the `send` can send a command back to the initiator while processing the script, without danger of deadlock. In addition, you can use the `-async` option with `send` to tell the `send` command to work asynchronously and not wait for a response to the commands sent.
One of the most common uses of `send` is to try simple experiments in Tk applications that are already running, such as changing a color or modifying the command associated with a button. Most Tk applications don't present an interface for typing Tcl commands directly to the application. However, you can always issue commands to such applications by starting an interactive `wish` application and then invoking `send`. For example, the following command changes the background color of a particular window in an application named `scan`:

```
send scan {.menubar.file configure -bg blue}
```

12.10.2 Application Names

To send to an application, you have to know its name. Each application on the display has a unique name, which it can choose in any way it pleases as long as it is unique. In many cases the application name is just the name of the program that created the application. For example, `wish` uses the application name `wish` by default; or, if it is running under the control of a script file, it uses the name of the script file as its application name. In programs like editors that are associated with a file or object, the application name typically has two parts: the name of the application and the name of the file or object on which it is operating. For example, if an editor named `mx` is displaying a file named `tk.h`, the application's name is likely to be `mx tk.h`.
If an application requests a name that is already in use, Tk adds a number to the end of the new name to keep it from conflicting with the existing name. For example, if you start up `wish` twice on the same display, the first instance will have the name `wish` and the second instance will have the name `wish #2`. Similarly, if you open a second editor window on the same file, it will end up with a name like `mx tk.h #2`.
Tk provides three commands that return information about the names of applications. First, the command

```
    winfo name .
⇒ wish #2
```

returns the name of the invoking application. Second, the command

```
    winfo interps
⇒ wish {wish #2} {mx tk.h}
```

returns a list whose elements are the names of all the applications defined on the display. Third, the command

```
selection get APPLICATION
```

returns the name of the Tk application that currently owns the selection; if no Tk application currently owns the selection, the command generates an error.

Note

Use the `tk appname` command to change an application name.

Use the `-displayof` option to specify a different X Window display. Pass the path to a window that Tk can then use to identify the display.

12.10.3 Security Issues with `send`

The `send` command is potentially a major security loophole. Any application that uses your display can send scripts to any Tk application on that display, and the scripts can use the full power of Tcl to read and write your files or invoke subprocesses with the authority of your account. Ultimately this security problem must be solved in the X display server, since even applications that don't use Tk can be tricked into abusing your account by sufficiently sophisticated applications on the same display. However, without Tk it is relatively difficult to create invasive applications; with Tk and `send` it is trivial.
You can protect yourself fairly well if you employ a key-based protection scheme for your display like `xauth` instead of a host-based scheme like `xhost`. `xauth` generates an obscure authorization string and tells the server not to allow an application to use the display unless it can produce the string. Typically the string is stored in a file that can be read only by a particular user, so this restricts use of the display to the one user.
In order to provide at least a small amount of security, Tk checks the access control being used by the server and rejects incoming sends unless `xhost`-style access control is enabled (that is, only certain hosts can establish connections) and the list of enabled hosts is empty. This means that applications cannot connect to your server unless they use some other form of authorization such as that provided by `xauth`.

13. Errors and Exceptions

A Tcl command can return errors for a variety of reasons, such as when it doesn't receive the right number of arguments, or if the arguments have the wrong form, or because some other problem occurs in executing the command, such as an error in a system call for file I/O. In most cases errors represent severe problems that make it impossible for the application to complete the script it is processing. Tcl's error facilities are intended to make it easy for the application to unwind the work in progress and display an error message to the user that indicates what went wrong. Presumably the user will fix the problem and retry the operation.
Errors are just one example of a more general phenomenon called *exceptions*. Exceptions are events that cause scripts to be aborted; they include the `break`, `continue`, and `return` commands as well as errors. Tcl allows exceptions to be "caught" by scripts so that only part of the work in progress is unwound. After catching an exception, the script can ignore it or take steps to recover from it. If the script can't recover, it can reissue the exception.

13.1 Commands Presented in This Chapter

The following Tcl commands are related to exceptions:

- `catch` *`command`* `?`*`returnVar`*`?` `?`*`optionsVar`*`?`

Evaluates *`command`* as a Tcl script and returns an integer code that identifies the completion status of the command. If *`returnVar`* is specified, it gives the name of a variable that will be set to the return value or error message generated by *`command`*. If *`optionsVar`* is specified, it gives the name of a variable that will be set to the return options dictionary.

- `error` *`message`* `?`*`info`*`?` `?`*`code`*`?`

Generates an error with *`message`* as the error message. If *`info`* is specified and is not an empty string, it is used to initialize the `errorInfo` variable. If *`code`* is specified, it is stored in the `errorCode` variable.

- `return ?`*`option value ...`*`? ?`*`result`*`?`

Causes the current procedure to return an exceptional condition. If used, `-code` specifies the return condition and its value must be `ok`, `error`, `return`, `break`, `continue`, or an integer. The `-errorinfo` option may be used to specify a starting value for the `errorInfo` variable, and `-errorcode` may be used to specify a value for the `errorCode` variable. *`result`* gives the return value or error message associated with the return; it defaults to an empty string. In support of constructing advanced, custom control structures, you can also provide any number of *`option value`* pairs, with arbitrary *`option`* names, which become entries in the return options dictionary. Also, you can include an explicit `-options` argument, whose value must be a valid dictionary, the entries of which become additional *`option value`* pairs in the return options dictionary.

- `interp bgerror` *`path`* `?`*`cmdPrefix`*`?`

Registers *`cmdPrefix`* as the background error handler for the interpreter specified by *`path`*. The *`cmdPrefix`* is a command name optionally followed by any number of arguments. When a background error occurs, the command is invoked with the specified arguments and two additional ones: an error message and a return options dictionary. If *`cmdPrefix`* is not provided, `interp bgerror` returns the command prefix currently registered for the interpreter.

13.2 What Happens after an Error?

When a Tcl error occurs, the Tcl interpreter aborts execution of the current command and raises an error condition. If the command is part of a larger script, the error condition also aborts the script execution. If the error occurs while a Tcl procedure is executing, the procedure is aborted, along with the procedure that called it, and so on until all the active procedures have aborted. After all Tcl activity has been unwound in this way, control eventually returns to the application executing the Tcl code, along with an indication that an error occurred and a message describing the error. It is up to the application to decide how to handle this situation, but most interactive applications, such as `tclsh` running in interactive mode, display the error message for the user and continue processing user input. In a batch-oriented application, such as when `tclsh` is invoked with the name of a script file to execute, applications often print the error message to the console and exit.
For example, consider the following script, which is intended to sum the elements of a list:

```
    set list {44 16 123 98 57}
    set sum 0
    foreach el $list {
        set sum [expr {$sum+$element}]
    }
Ø can't read "element": no such variable
```

This script is incorrect because there is no variable `element`: the variable name `element` in the `expr` command should have been `el` to match the loop variable for the `foreach` command. When the script is executed, an error occurs as Tcl parses the `expr` command: Tcl attempts to substitute the value of variable `element` but can't find a variable by that name, so it signals an error. This error indication is returned to the `foreach` command, which invoked the Tcl interpreter to evaluate the loop body. When `foreach` sees that an error has occurred, it aborts its loop and returns the same error indication as its own result. This in turn aborts the overall script. The error message

```
can't read "element": no such variable
```

is returned along with the error condition, which is displayed for the user.
In many cases the error message provides enough information for you to fix the problem. However, if the error occurred in a deeply nested set of procedure calls, you might not be able to figure out where it occurred from the message alone. To help pinpoint the location of the error, Tcl creates a stack trace as it unwinds the commands that were in progress, and it stores the stack trace in the global variable `errorInfo`. The stack trace describes each of the nested calls to the Tcl interpreter. For example, after the preceding error, `errorInfo` contains the following value:

```
can't read "element": no such variable
    while executing
"expr {$sum+$element}"
    ("foreach" body line 2)
    invoked from within
"foreach el $list {
    set sum [expr {$sum+$element}]
}"
```

Tcl provides one other piece of information about error conditions in the global variable `errorCode`. `errorCode` has a format that is easy to process with Tcl scripts; it is most commonly used in Tcl scripts that attempt to recover from errors using the `catch` command, described later. The `errorCode` variable consists of a list with one or more elements. The first element identifies a general class of errors, and the remaining elements provide more information in a class-dependent fashion. For example, if the first element of `errorCode` is `POSIX`, it means that an error occurred in a POSIX system call. `errorCode` then contains two additional elements giving the POSIX name for the error, such as `ENOENT`, and a human-readable message describing the error.

Other error codes indicate arithmetic, file access, and child process exceptions. Not all Tcl commands set the error code explicitly. If a command generates an error without setting `errorCode`, Tcl fills it in with the value `NONE`. See the reference documentation for a complete description of all the forms `errorCode` can take.

13.3 Generating Errors from Tcl Scripts

Tcl errors may be generated by the C code that implements the Tcl interpreter and built-in commands, or by scripts using the Tcl `error` command as in the following example:

```
if {($x < 0) || ($x > 100)} {
    error "x is out of range ($x)"
}
```

The `error` command generates an error and uses its argument as the error message.
As a matter of programming style, you should use the `error` command only in situations where the correct action is to abort the script being executed. If you think that an error is likely to be recovered from without aborting the entire script, it is probably better to use the normal return value mechanism to indicate success or failure (e.g., return one value from a command if it succeeded and another if it failed, or set variables to indicate success or failure). Although it is possible to recover from errors (you'll see how in Section 13.4), the recovery mechanism is more complicated than the normal return value mechanism. Thus, you should generate errors only when recovery is unlikely.

13.4 Trapping Errors with `catch`

Errors generally cause all active Tcl commands to abort execution, but there are some situations where it is useful to continue executing a script after an error has occurred. For example, suppose that you want to use the `open` command to open a file to read its contents. (Tcl commands for manipulating files are discussed in Chapter 11.) Although Tcl provides the `file readable` command to test whether a file exists and your process has permission to read its contents, another process on your system could delete the file before you get the opportunity to open it. If the file does not exist, `open` generates an error:

```
    open msg.txt
∅ couldn't open "msg.txt": no such file or directory
```

You can use the `catch` command to ignore the error in this situation:

```
    catch {open msg.txt}
⇒ 1
```

The argument to `catch` is a Tcl script, which `catch` evaluates. If the script completes normally, `catch` returns `0`. If an error occurs in the script, `catch` traps the error (so that the `catch` command itself is not aborted by the error) and returns `1` to indicate that an error occurred. This example ignores any errors in `open`. However, we would still need to test the return value of `catch` to determine if the `open` was successful or not.
The `catch` command can also take a second argument. If the argument is provided, it is the name of a variable, and `catch` modifies the variable to hold either the script's return value (if it returns normally) or the error message (if the script generates an error):

```
    catch {open msg.txt} fid
⇒ 1
    set fid
⇒ couldn't open "msg.txt": no such file or directory
```

In this case, the `open` command generates an error, so `fid` is set to contain the error message. If the file had existed, `open` would have returned successfully, so the return value from `catch` would have been `0` and `fid` would have contained the return value from the `open` command, which is the channel identifier of the file opened. This longer form of `catch` is useful if you need access to the return value when the script completes successfully. It's also useful if you need to do something with the error message after an error, such as logging it to a file.
You can also provide a variable name as an optional third argument to capture the return options dictionary, which is described in the next section.

13.5 Exceptions in General

Errors are not the only things in Tcl that cause work in progress to be aborted. Errors are just one example of a set of events called *exceptions*. In addition to errors, there are three other kinds of exceptions in Tcl, which are generated by the `break`, `continue`, and `return` commands. All exceptions cause active scripts to be aborted in the same way, except for two differences. First, the `errorInfo` and `errorCode` variables are set only during error exceptions. Second, the exceptions other than errors are almost always caught by an enclosing command, whereas errors usually unwind all the work in progress. For example, the `break` and `continue` commands are normally invoked inside a looping command such as `foreach`; `foreach` catches `break` and `continue` exceptions and terminates the loop or skips to the next iteration. Similarly, `return` is normally invoked only inside a procedure or a file being `source`d. Both the procedure implementation and the `source` command catch return exceptions.

Note

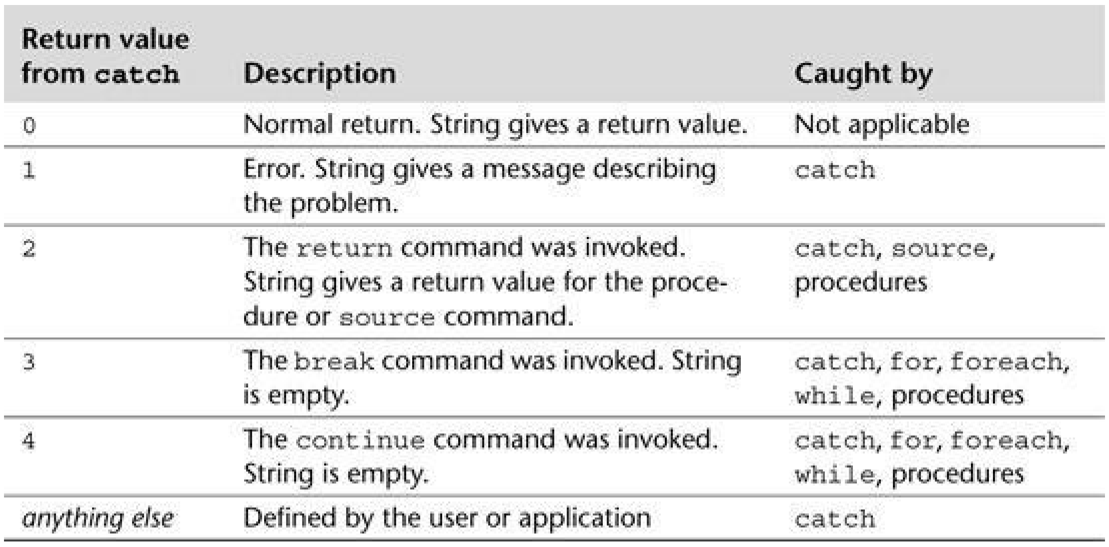
If a `break` or `continue` command is invoked outside any loop, active scripts are unwound until the outermost script for a procedure is reached or all scripts in progress have been unwound. At this point Tcl turns the `break` or `continue` exception into an error with an appropriate message.

All exceptions are accompanied by a string value. In the case of an error, the string is the error message. In the case of `return`, the string is the return value for the procedure or script. In the case of `break` and `continue`, the string is always empty.

The `catch` command actually catches all exceptions, not just errors. The return value from `catch` indicates what kind of exception occurred, and the variable specified in `catch`'s second argument is set to hold the string associated with the exception.

Table 13.1 describes the standard exception types. The first column indicates the value returned by `catch` in each instance. The second column describes when the exception occurs and the meaning of the string associated with the exception. The last column lists the commands that catch exceptions of that type ("procedures" means that the exception is caught by a Tcl procedure when its entire body has been aborted). The top row refers to normal returns where there is no exception.

Table 13.1 Summary of Tcl Exceptions



Return value from `catch`	Description	Caught by
0	Normal return. String gives a return value.	Not applicable
1	Error. String gives a message describing the problem.	`catch`
2	The `return` command was invoked. String gives a return value for the procedure or `source` command.	`catch`, `source`, procedures
3	The `break` command was invoked. String is empty.	`catch`, `for`, `foreach`, `while`, procedures
4	The `continue` command was invoked. String is empty.	`catch`, `for`, `foreach`, `while`, procedures
anything else	Defined by the user or application	`catch`

In the following example, the `catch` command has a return value of 2, which indicates that it encountered a return condition while executing its script argument. `catch` stores the string associated with the exception in the variable `string`.

```
  catch {return "all done"} string
⇒ 2
  set string
⇒ all done
```

Whereas `catch` provides a general mechanism for catching exceptions of all types, `return` provides a general mechanism for generating exceptions of all types. If its first argument consists of the keyword `-code`, as in

```
return -code return 42
```

its second argument is the name of an exception (`return` in this case), and the third argument is the string associated with the exception. The enclosing procedure returns immediately, but instead of a normal return it returns with the exception described by the `return` command's arguments. In the preceding example, the procedure generates a return exception, which then causes the calling procedure to return as well.
Additionally, the `return` command allows any number of option-value pairs, where you can select any name desired for the options. Some option names, which are listed in Table 13.2, receive special treatment. All option-value pairs become entries in the *return options dictionary*, which the `catch` command can capture by specifying a variable name as an optional third argument. This allows you to pass any arbitrary information desired when

raising an exception. Typically this feature is used only when implementing advanced control structures.

Table 13.2 Significant Option Names for the `return` Command

return option name	**Description**
`-code` *`code`*	The exception code, as discussed in Table 13.1.
`-errorcode` *`list`*	If the exception code is `error`, Tcl sets the value of the `errorCode` variable to *`list`* if this option is present, `NONE` otherwise.
`-errorinfo` *`info`*	If the exception code is `error`, Tcl sets the initial value of the `errorInfo` variable to *`info`* if this option is present, a generated stack trace otherwise.
`-level` *`level`*	The *`level`* value must be a non-negative integer. It specifies a command currently executing *`level`* levels up the stack, whose return code will be set to the value given by `-code`. The default is `1`.
`-options` *`options`*	The value *`options`* must be a valid dictionary. The entries of that dictionary are treated as additional option-value pairs for the return options dictionary.

In Section 9.6 you saw how a new looping command, `do`, could be implemented as a Tcl procedure using `upvar` and `uplevel`. However, the example in Section 9.6 did not properly handle exceptions within the loop body. Here is a new implementation of `do` that uses `catch` and `return` to deal with exceptions properly:

```
proc do {varName first last body} {
    upvar 1 $varName v
    for {set v $first} {$v <= $last} {incr v} {
        set code [catch {uplevel 1 $body} string options]
        switch -- $code {
            0 -
            4         {}
            3         {return}
            default   {
                    dict incr options -level
                    return -options $options $string
            }
        }
    }
}
```

This new implementation handles exceptions in much the same way as built-in looping commands such as `foreach` and `while`. It evaluates the loop body

inside a `catch` command and then checks to see how the body terminates. If no exception occurs (`code` is `0`) or if the exception is a continue (`code` is `4`), `do` goes on to the next iteration. If a break exception occurs (`code` is `3`), `do` returns to its caller normally, ending the loop. If an error, return, or any other completion code occurs, `do` simply reflects that exception back to its caller.
When `do` reflects an exception to its caller, for the most part all it needs to do is pass the entire return options dictionary to its caller. The only change it needs to make in the return options dictionary is to increment the value of the `-level` option by 1. In typical execution, this causes the exception to appear in the context of caller, rather than in the context of the `do` procedure. But it also handles situations properly where the exception had its `-level` set explicitly in the body of code executed.

13.6 Background Errors and `bgerror`

A *background error* is one that occurs in an event handler. For example, if an error occurs while executing a command specified with the `after` command or the `-command` option of a widget, it is a background error. For a non-background error, the error simply can be returned up through nested Tcl command evaluations until it reaches the top-level code in the application or is caught and handled with a `catch` command, as described in previous sections. On the other hand, when a background error occurs, the unwinding ends in the Tcl library and there is no obvious way for Tcl to report the error.
When a Tcl interpreter detects a background error, it saves information about the error and invokes a handler command. You can register a background error handler with the `interp bgerror` command; each interpreter in your application can have its own background error handler. If you don’t explicitly register a background error handler, Tcl uses a default handler.
For compatibility with versions of Tcl prior to 8.5, the default handler checks to see whether a command called `bgerror` is defined in the interpreter. If so, it invokes `bgerror` and passes a single argument consisting of the error message generated; the global `errorCode` and `errorInfo` variables are also set to their values at the time the error occurred. If you have not registered a `bgerror` command, or an error occurs during the execution of `bgerror`, Tcl reports the error itself by writing a message to the `stderr` channel.
Starting with Tcl 8.5, you can use `interp bgerror` to register more advanced background error handlers. The first argument to `interp bgerror` is a path to the interpreter, as defined in Chapter 15; the path `{}` refers to the current

interpreter. Following that, you can provide additional arguments specifying a command and any explicit arguments that you want to pass to that command when invoked as the background error handler. When the command is invoked, two additional arguments are provided, consisting of the error message and a return options dictionary for the error, as discussed in Section 13.5. The result is executed in the interpreter's global namespace. Executing `interp bgerror` with no arguments following the interpreter path returns the command prefix currently registered as the background error handler.

The following demonstrates an example of registering a background error handler that simply sends the error message and return options dictionary to the standard error channel:

```
  proc myHandler {errMsg returnOpts} {
      puts stderr "Background error: $errMsg"
      puts stderr "Return options dictionary:"
      dict for {key value} $returnOpts {
          puts stderr "  $key: $value"
      }
  }
  interp bgerror {} myHandler
  after 0 { expr 1/0 }
  update
⇒ Background error: divide by zero
  Return options dictionary:
    -code: 1
    -level: 0
    -errorcode: ARITH DIVZERO {divide by zero}
    -errorinfo: divide by zero
    invoked from within
  "expr 1/0 "
      ("after" script)
    -errorline: 1
```

14. Creating and Using Tcl Script Libraries

Like any other modern programming language, Tcl allows you to group commonly used procedure definitions into *libraries* that can be used by many applications. A Tcl library can be as simple as a single script file defining a few utility procedures. Or you can combine multiple library script files into a single *package* that you can distribute. Packages also have version numbers associated with them, so applications can distinguish among different historic versions of a package and require specific ones.
Additionally, Tcl allows you to bundle entire Tcl applications or libraries into single-file packs called *Starkits*. Combined with the *Tclkit* single-file Tcl interpreter, this gives you a handy means of distributing your Tcl applications. You can even create a single-file application executable called a *Starpack*, which combines a Tclkit for a particular platform with a Starkit of your software.
Modern extensions should be implemented as *modules* or packages and make use of *autoloading*, where the Tcl interpreter automatically loads modules or packages as needed. To support autoloading, the Tcl interpreter makes use of a few simple conventions, which grew out of the libraries that shipped with early versions of Tcl.
The following sections show how modern packages and modules are built on top of Tcl's support for libraries of procedures, automatically loading scripts as needed. The final section shows how you can use Tclkit and Starkits or Starpacks to distribute your applications easily on multiple target platforms.

14.1 Commands Presented in This Chapter

- `auto_mkindex` *`dir`* `?`*`pattern ...`*`?`

Generates an index suitable for use by Tcl's autoloading mechanism. The command searches *`dir`* for all files whose names match any of the *`pattern`* arguments (matching is done with the `glob` command), generates an index of all the Tcl command procedures defined in all the matching files, and stores the index information in a file named `tclIndex` in *`dir`*. If no *`pattern`* is given, a pattern of `*.tcl` is assumed.

- `info library`

Returns the name of the library directory in which standard Tcl scripts are stored. This is actually the value of the `tcl_library` global variable and may be changed by setting `tcl_library`.

- `info loaded ?`*`interp`*`?`

Returns a list describing all of the packages that have been loaded into *`interp`* with the `load` command. Each list element is a sublist with two elements, consisting of the name of the file from which the package was loaded and the name of the package. For statically loaded packages the file name is an empty string. If *`interp`* is omitted, information is returned for all packages loaded in any interpreter in the process. To get a list of just the packages in the current interpreter, specify an empty string for the *`interp`* argument.

- `info sharedlibextension`

Returns the file extension used on this platform for shared libraries.

- `package ifneeded` *`package version script`*

Used in a `pkgIndex.tcl` file to indicate that the specified *`version`* of a *`package`* is available if needed. The optional *`script`* argument executed automatically by Tcl in the *`package`* is later requested with a package `require` command.

- `package names`

Returns a list of the names of all packages in the interpreter for which a version has been provided (via `package provide`) or for which a `package ifneeded` script is available.

- `package prefer ?latest|stable?`

Sets which type of package is preferred: `latest`, which matches the latest version of a package present even if it is an alpha or beta release; or `stable`, which matches an alpha or beta version of a package only if no stable version fulfills the requirements. Once an interpreter's preference is set to `latest`, attempts to set it back to `stable` are ignored. Without an argument, `package prefer` returns the current preference.

- `package provide` *`package`* `?`*`version`*`?`

Indicates that version *`version`* of package *`package`* is now present in the interpreter. It is typically invoked once as part of a `package ifneeded` script and again by the package itself when it is finally loaded. If the version argument is omitted, the command returns the version number that is currently provided, or an empty string if no `package provide` command has been invoked for *`package`* in this interpreter.

- `package require` *`package`* `?`*`requirement`*`?`

  `package require -exact` *`package requirement`*

Loads the *`package`* specified, ensuring that the version *`requirement`* is met. The first version loads the latest version of *`package`* available on the system, as long as it meets the minimum version *`requirement`* and its major version number is the same as *`requirement`*. With the `-exact` option, only the version of *`package`* specified by *`requirement`* is accepted. The command returns the version of *`package`* loaded, or an error if a *`package`* meeting *`requirement`* is not available.

- `package vcompare` *`version1 version2`*

Compares the two version numbers given by *`version1`* and *`version2`*. Returns `-1` if *`version1`* is an earlier version than *`version2`*, `0` if they are equal, and `1` if *`version1`* is later than *`version2`*.

- `package versions` *`package`*

Returns a list of all the version numbers of *`package`* for which information has been provided by `package ifneeded` commands.

- `pkg_mkIndex ?`*`options`*`?` *`dir`* `?`*`pattern`* `...?`

Generates a `pkgIndex.tcl` index suitable for use by Tcl's package mechanism. The command searches *`dir`* for all files whose names match any of the *`pattern`* arguments (matching is done with the `glob` command), generates an index of all the Tcl command procedures defined in all the matching files, and stores the index information in a file named `pkgIndex.tcl` in *`dir`*. If no *`pattern`* is given, the default patterns of `*.tcl` and `*.[info sharedlibextension]` are used. See the reference documentation and Section 14.5.2 for more information on the supported *`options`*.

- `::tcl::tm::path list`

Returns a list containing all registered Tcl module paths, in the order in which they are searched for modules.

14.2 The `load` Command

The `load` command allows you to load precompiled libraries, typically written in C or C++, into the Tcl interpreter. To do so, your binary file must be created as a shared library on Unix or a Dynamic Link Library (DLL) on Windows. Because different platforms use different extensions to identify shared library files, Tcl provides the `info sharedlibextension` command, which returns the shared library extension used by the current system. Thus, a shared library is often loaded like this:

```
load myextension[info sharedlibextension]
```

Your C or C++ code must also follow certain Tcl conventions. Once the shared library implementing the package is loaded, the Tcl interpreter calls

the *initialization* function for the package. The name of this function is based on the name of the package. For example, with a package named `star1`, the initialization method would be `Star1_Init`. If the binary file is loaded by a safe Tcl interpreter, the initialization function name would be `Star1_SafeInit`. See Chapter 36 for more information on writing packages in C.
Once you load a binary file, you can use the `info loaded` command to list all packages loaded with the `load` command.

14.3 Using Libraries

A Tcl library is a directory containing one or more Tcl script files implementing a related set of procedures. Tcl ships with a standard library of procedures that implement some of its default behavior. Understanding how the Tcl library works is a good way to learn how to add your own libraries and packages.
The command `info library` returns the full path name of the Tcl library directory, such as

```
info library
⇒ C:/Program Files/Tcl8.5/lib/tcl8.5
```

In addition, the global variable `tcl_library` holds the same value:

```
puts $tcl_library
⇒ C:/Program Files/Tcl8.5/lib/tcl8.5
```

This directory is used to hold standard scripts used by Tcl, such as a default definition for the `unknown` procedure described next.

14.4 Autoloading

Section 15.10 describes how Tcl handles processing unknown commands by invoking a procedure called `unknown`. Tcl has a default implementation of the `unknown` command, but you can define your own to implement custom functionality. One of the most useful functions performed by the default `unknown` procedure is *autoloading*. Autoloading allows you to write collections of Tcl procedures and place them in script files in library directories. You can then use these procedures in your Tcl applications without having to `source` the files that define them explicitly; you simply invoke the procedures. The first time that you invoke a library procedure it doesn't exist, so `unknown` is called. `unknown` finds the file that defines the procedure, `source`s the file to define the procedure, and then re-invokes the original command. The next time the procedure is invoked it exists, so the autoloading mechanism isn't triggered.

Autoloading provides two benefits. First, it makes it easy to build large libraries of useful procedures and use them in Tcl scripts. You need not know exactly which files to `source` to define which procedures, since the autoloader takes care of that for you. The second benefit of autoloading is efficiency. Without autoloading, an application must `source` all of its script files when it starts. Autoloading allows an application to start up without loading any script files at all; the files are loaded later when their procedures are needed, and some files may never be loaded at all. Thus autoloading reduces startup time and saves memory.

The autoloader is straightforward to use. First, create a library as a set of script files in a single directory. Normally these files have names that end in `.tcl`, for example, `db.tcl` or `stretch.tcl`. Each file can contain any number of procedure definitions. It's a good practice to keep the files relatively small, with just a few related procedures in each one. In order for the autoloader to handle the files properly, the `proc` command for each procedure definition must be at the beginning of a line with no leading space, and it must be followed immediately by whitespace and the procedure's name on the same line. Other than this, the format of the script files doesn't matter as long as they are valid Tcl scripts.

The second step in using the autoloader is to build an index. To do this, start a Tcl application such as `tclsh` and invoke the `auto_mkindex` command as in the following example:

```
auto_mkindex . *.tcl
```

`auto_mkindex` isn't a built-in command but rather a procedure in Tcl's script library. Its first argument is a directory name, and the second argument is a glob-style pattern that selects one or more script files in the directory. `auto_mkindex` scans all of the files whose names match the pattern and builds an index that indicates which procedures are defined in which files. It stores the index in a file called `tclIndex` in the directory. If you modify the files to add or delete procedures, you should regenerate the index.

The final step is to set the variable `auto_path` in the applications that wish to use the library. The `auto_path` variable contains a list of directory names.

When the autoloader is invoked, it searches the directories in `auto_path` in order, looking in their `tclIndex` files for the desired procedure. If the same procedure is defined in several libraries, the autoloader uses the one from the earliest directory in `auto_path`. Typically, `auto_path` is set as part of an application's startup script. For example, if an application uses a library in the directory `/usr/local/lib/shapes`, it might include the following command in its startup script:

```
set auto_path \
        [linsert $auto_path 0 /usr/local/lib/shapes]
```

This adds `/usr/local/lib/shapes` to the beginning of the path, retaining all the existing directories in the path such as those for the Tcl and Tk script libraries but giving higher priority to procedures defined in `/usr/local/lib/shapes`. Once a directory has been properly indexed and added to `auto_path`, all of its procedures become available through autoloading.

Note

More complex libraries—especially those involving an integrated mixture of Tcl scripts and C extensions, or where there is a need to maintain multiple versions of a library—are better implemented as packages. Packages are also strongly preferred when libraries are distributed to other people, since they are considerably easier to manage and install.

14.5 Packages

Creating libraries gives you the ability to collect useful utility procedures and build libraries of code for reuse. With autoloading, your Tcl programs gain efficiencies by loading only what is needed. Even so, Tcl libraries don't provide a number of useful features such as versioning and better code separation. For example, if you have two procedures with the same name, the first one found in the library path, `auto_path`, gets called.

Note

You can alleviate some of this problem by using namespaces, covered in Chapter 10.

By creating a Tcl package, you can add versioning and provide a better organization to your library code.

14.5.1 Using Packages

Tcl already comes with a number of packages, but you can also download many more. (Appendix B describes some of the popular Tcl packages available that extend Tcl's functionality.) To use a package, specify the package name and version with the `package require` command; for example:

```
package require platform 1.0.3
```

This command attempts to load the required package. Once it is loaded, you can call on the functionality of the package within your Tcl program.
By convention, versions with the same major number are considered compatible; that is, if your code asks for `platform` at version 1.0, then 1.0.3, 1.1, 1.2, or other higher minor versions would satisfy the requirement. In this case, a package at version 1.0.4 would satisfy the requirement, but 1.0.2 would not. In addition, a version of 2.0 would not satisfy the requirement. The Tcl package mechanism selects the highest-numbered compatible version of the package that satisfies the requirement.
If you are not concerned with the version number, use a command like the following:

```
package require platform
```

This command loads the highest version of the package installed on your system. Or, if you require an exact version, use the `-exact` option:

```
package require -exact platform 1.0.3
```

Note

Version numbers can also include an *a* for alpha or a *b* for beta, such as

1.3b1. These letters indicate unstable packages. Package versions without letters indicate stable versions. Tcl normally prefers stable over unstable versions. You can change the preference to load the latest available version of a package, even if it is alpha or beta, with the command

```
package prefer latest
```

14.5.2 Creating Packages

A Tcl package is a directory of one or more Tcl scripts and/or binary shared libraries. Each Tcl script in the package should declare that it provides the package with the `package provide` command; for example:

```
package provide platform 1.0.3
```

This command specifies that the source in the file provides (or helps to provide if there is more than one Tcl script in the directory) a package named `platform`. The version number is 1.0.3. with the major number of 1 and a minor number of 0.3. Missing numbers are treated as zeros, so 1.0.3 is the same as 1.0.3.0 and 1.0.3.0.0.0. You can make a long version number if needed, such as 1.0.3.0.3.0.3.0.3.0.3, though this is not necessary in most cases. You might also use a build number or a timestamp as part of the version number, such as 1.1.20080829.

Note

The first number, the major number, is the most important. You should increment your package's major version number whenever you make a change that is not backward-compatible with previous versions.

When you create a package, place all your code into a single directory. Then you need to create a `pkgIndex.tcl` file, which the Tcl interpreter uses to load your package. You can create this file by hand, and if you have a complex case you'll need to do so. But most packages can use the handy `pkg_mkIndex` procedure. `pkg_mkIndex` creates the index file needed by the Tcl package system to manage loading your package. As with `auto_mkindex`, `pkg_mkIndex` isn't a built-in command but rather a procedure in Tcl's script library.

Note

Even if you write your own `pkgIndex.tcl` script, you can use the `pkg_mkIndex` procedure as a guide that defines what the Tcl interpreter expects in a `pkgIndex.tcl` script.

The basic format of the command is

```
pkg_mkIndex . *.tcl
```

This command looks at all files with names ending in `.tcl` in the current directory and creates the package index in the file `pkgIndex.tcl` in the directory indexed. The `pkgIndex.tcl` holds one or more `package ifneeded` commands, along with the scripts needed to load the packages; for example:

```
package ifneeded app-star1 1.0 \
    [list source [file join $dir star1.tcl]]
```

This command specifies a package named `app-star1` at version `1.0`. The command to load the package is to `source` in the Tcl script file, `star1.tcl`. The value `$dir` gets passed in by the Tcl interpreter when loading the package. The first time a `package require` command gets executed, the Tcl interpreter executes the `package unknown` script. This script `source`s all the `pkgIndex.tcl` files found in the directories listed in the `auto_path` global variable and their immediate subdirectories.

Note

Any platform-specific binary files must be set up to be loaded with the `load` command, described in Section 14.2. In addition, the binary file must contain a call to the C function `Tcl_PkgProvide`.

`pkg_mkIndex` supports a number of options, including `-direct`, to specify loading the package files when a `package require` command is executed, which is the default. The `-lazy` option is discouraged, but it sets up the package to get loaded only when one of the procedures in the package is first called. The `-verbose` option tells `pkg_mkIndex` to output status information as it works. A special option, `-load`, specifies which packages to preload using a pattern. This is to help index packages containing binary files that in turn depend on other packages.

14.5.3 Using `::pkg::create`

The `::pkg::create` procedure creates the `package ifneeded` command used in the `pkgIndex.tcl` files. You can also use `::pkg::create` when creating your `pkgIndex.tcl` files by hand. Normally you can simply use `pkg_mkIndex` to create a package index script. The basic syntax is

```
::pkg::create -name package_name \
      -version version_number -source tcl_scripts
```

The `-source` parameter takes a list with the first element being a file name and the second an optional list of commands provided in the file; for example:

```
  ::pkg::create -name foo -version 99.5 -source star1.tcl
⇒ package ifneeded foo 99.5 [list source [file join $dir star.tcl]]
```

If you have a binary file, use the `-load` option:

```
::pkg::create -name package_name \
    -version version_number \
    -load {binary_file commands_provided}
```

The `-load` parameter takes a list whose first element is a file name and the second is an optional list of commands provided in the binary file.

14.5.4 Installing Packages

To install your new package, install the package's directory as a subdirectory of any directory listed in the `::tcl_pkgPath` variable. Once your package is installed, the `package require` command can find it.

> **Note**
>
> By convention, if the `::tcl_pkgPath` variable contains more than one directory, you should place platform-specific packages into the first directory in the list. Place platform-agnostic packages in the second

directory in the list.

If you need to install your package in another location (that is, not under the directories listed in the `::tcl_pkgPath` variable)—or if the `::tcl_pkgPath` variable is not set on your system—you must place your directory under one of the directories listed in the `::auto_path` variable. Typically, the directory in which Tcl looks for its initialization scripts is one of the directories listed in the `::auto_path` variable; the `info library` command returns the name of this directory. When in doubt, this directory is often a reasonable target directory for installing your package.
You can also add the directory name where you installed your package to the `::auto_path` variable. You need to do this only if you install your package to a nonstandard location.

14.5.5 Utility Package Commands

The `package names` command lists all packages installed:

```
  package names
⇒ activestate::teapot::link http platform tcl::tommath tcltest
  msgcat ActiveTcl Tcl
```

Note

This example shows packages installed with ActiveTcl.

The `package versions` command returns a list of all versions of a package that are available on the system:

```
  package versions mypackage
⇒ 1.0 1.1
```

When the `package provide` command is executed without a version number, it returns the version number that is currently provided, or an empty string if no `package provide` command has been invoked for the package in this interpreter:

```
  package provide Trf
⇒ 2.1.2
```

14.6 Tcl Modules

The historical mechanism for locating and loading packages employed by the Tcl core is very flexible, but it suffers from drawbacks as well. The primary problems with the historical mechanism are that it extensively searches the file system for packages, and that it has to actually read a file (`pkgIndex.tcl`) to get the full information for a prospective package. All of these operations take time. The fact that “index scripts” are able to extend the list of paths searched tends to heighten this cost, as it forces rescans of the file system. Installations where directories in the `auto_path` are large or mounted from remote hosts are hit especially hard by this (especially network delays). All of this together causes a slow startup of `tclsh` and Tcl-based applications, especially for systems with an extensive collection of packages located on remote hosts.
Tcl 8.5 added support for *Tcl modules*, which are less flexible than traditional packages but require far less file system access by the Tcl interpreter when it is searching for available packages and modules. Tcl modules are the recommended method for creating and distributing extensions in Tcl 8.5 and beyond.
The main simplification is that each module must be stored in a single file, a file that is read with the Tcl `source` command. Inside the file, the module may define one or more Tcl packages, as described in Section 14.5. Typically, each module file defines a single Tcl package. You can store binary data within the same file after a `Control-z`, which ends the reading of Tcl commands. The other simplification is that there is no index file for the module (that is, you don’t create a `tclIndex` or `pkgIndex.tcl` file to accompany the module). The name of the module file and its location are enough to tell the Tcl interpreter what package and version the module provides.

Note

For more information on creating modules containing binary data, embedded ZIP archives, and other more complex structures, see the documentation for TIP 190 at `http://tip.tcl.tk/189.html` and search for related articles on the Tcler’s Wiki at `http://wiki.tcl.tk/`.

14.6.1 Using Tcl Modules

To use packages defined in Tcl modules, place a `package require` command within your Tcl program, just as for any package; for example:

```
package require platform::shell 1.1.3
```

The Tcl interpreter looks first for packages within Tcl module files. Tcl searches a list of directories known as the *module path*, which is completely independent of the `auto_path` list of directories described in Section 14.5.4. Only if an appropriate Tcl module is not found does Tcl fall back to using the traditional method of scanning the directories listed in `auto_path` for a package meeting the requirements.

14.6.2 Installing Tcl Modules

When creating modules, you must store them in a file whose name matches the regular expression

```
([[:alpha:]][:[:alnum:]]*)-([[:digit:]].*)\.tm
```

The first part of the file name specifies the name of the package, such as `myutils`. The portion of the file name following the `-` is the version number. Module file names must end in a `.tm` extension. For example, the following are valid module file names:

```
tcltest-2.3.0.tm
http-2.7.tm
shell-1.1.3.tm
myutils-1.1.20080829.tm
```

The directory tree for storing Tcl modules is independent of `auto_path`. Tcl modules are searched for in all directories listed in the result of the command `::tcl::tm::path list`. This is called the module path. Neither the `auto_path` nor the `tcl_pkgPath` variable is used.
When an application requests a package, before the search is started, the name of the requested package is translated into a partial path, where all occurrences of `::` in the package name are replaced by the appropriate

directory separator character for the platform. Thus on Unix each `::` in the requested package name is translated to a `/`. For example,

```
package require encoding::base64
```

results in a partial path of `encoding/base64`. After this translation, Tcl joins this partial path to the end of each directory in Tcl's module path list—which consists of a set of directories where Tcl looks for modules—to form a set of search patterns. The patterns are tested in the order of the module paths to search for a matching module file.

The default module path list is computed by `tclsh` based on the major and minor versions of the Tcl interpreter. So, using *X* to indicate the major version and *Y* to indicate each minor version less than or equal to the current minor version, the default path list consists of the following:

- `file normalize [info library]/../tclX/X. Y`

  `file normalize [info nameofexecutable]/../lib/tclX/ X. Y`

Thus, a version 8.5 Tcl interpreter would include a set of directories `tcl8/8.5`, `tcl8/8.4`, `tcl8/8.3`, `tcl8/8.2`, `tcl8/8.1`, and `tcl8/8.0`, where the `tcl8` directory is at the same level as the Tcl library directory returned by `info library`.

- `file normalize [info library]/../tclX/site-tcl`

Note that this is always a single entry, because *X* is the specific current major Tcl version.

- `$::env(TCLX_Y_TM_PATH)`

A list of paths, separated by either `:` on Unix or `;` on Windows. Note that *X* and *Y* follow the general rules mentioned above, so for Tcl 8.5, this would mean that six different environment variables would be looked for: `TCL8_5_TM_PATH` through `TCL8_0_TM_PATH`.

The command `::tcl::tm::path list` returns the current module path list. You can add directories to the list and remove them by using `::tcl::tm::path add` and `::tcl::tm::path remove`. See the `tm` reference documentation for more information.

As an example, consider the case where you've created the first version of a package that people would load into their application with the command

```
package require struct::btree
```

Further assume that you designed the package using features of Tcl 8.4, and it does not require any 8.5 features. If this were a module for your own personal use, you might put it in a subdirectory of your home directory. Assuming that your home directory is `/Users/ken`, the complete path name for the module might be

```
/Users/ken/modules/struct/btree-1.0.tm
```

You could create an environment variable named `TCL8_4_TM_PATH` and include the directory `/Users/ken/modules` as one of its values.
On the other hand, if your module were for general use by all users of your system, and Tcl is installed such that `info library` returns `C:\Tcl8.5\lib`, the complete path name for the module might be

```
C:\Tcl8.5\lib\tcl8\site-tcl\struct\btree-1.0.tm
```

If you plan to distribute the module, you could create a simple installation script that would query the `info library` command, and then install your module as

```
file normalize [info library]/../tcl8/8.4/struct/btree-1.0.tm
```

14.7 Packaging Your Scripts as Starkits

A Starkit provides a single-file distribution of your Tcl procedures, application data, and, if needed, platform-specific compiled code. This allows you to bundle an entire Tcl application as a single file.

Note

The name comes from an abbreviation for “stand-alone runtime.”

The Starkit itself uses Tcl’s virtual file system support to present your application with a common directory structure for locating code, application data, and other files. The files within the Starkit appear to your application as normal files, shielding your application from the bundling format.
To run the application within a Starkit, you need to use a special Tcl interpreter, called a Tclkit. A Tclkit is a platform-specific Tcl interpreter, built with a known set of Tcl extensions—extensions needed to extract and run your application from the virtual file system used by the Starkit. While a normal Tcl installation consists of hundreds of files, with a Tclkit you get a single-file executable with the entire Tcl distribution.

Note

As a single-file distribution mechanism, a Starkit is similar to a Java Jar file, and Tclkit is similar to the Java Runtime Engine, or JRE. The main difference is that you execute your Starkit directly using the Tclkit interpreter for a particular platform.

The Tclkit/Starkit combination mostly provides convenience for bundling and distribution—convenience for the end users of your Tcl programs. Without a facility like Tclkit and Starkits, you need several items to be able to deploy a Tcl program:

- A Tcl interpreter
- Platform-specific compiled libraries
- Tcl scripts that make up the Tcl standard library
- Any Tcl extensions needed by your application
- Any Tcl library code not part of the standard distribution but needed by your application
- Your application scripts
- Any platform-specific compiled code needed by your application
- Any data files needed by your application

The Starkit system divides these items into two bundles. The Tclkit provides the Tcl interpreter and a set of Tcl extensions, including everything necessary to run Starkit applications. The Starkit, which you create, contains your application scripts, any platform-specific compiled code needed, and your application data.

The next sections show how to create and work with Starkits. To create a Starkit, you first need to install a Tclkit interpreter.

14.7.1 Installing a Tclkit

Download a Tclkit interpreter for your platform from http://www.equi4.com/tclkit/download.html. Each Tclkit interpreter includes Tcl, Tk, IncrTcl, Metakit (a small embedded database library), and TclVFS support for virtual file systems.

Note

Find out more about Tclkits at `http://www.equi4.com/tclkit/`.

Normally, you just need to uncompress the downloaded file and you have a complete Tcl interpreter. You can invoke the Tclkit interpreter just as you would a `tclsh` and use it to run normal Tcl script applications as well as Starkit applications. Running a Starkit with a Tclkit is much like running a standard Tcl script:

```
tclkit starkitfile ?arguments_to_starkit ...?
```

14.7.2 Creating Starkits

The next step to creating Starkits is to download the `sdx` Starkit. `sdx` is a Starkit designed to help you create your own Starkits, allowing you to wrap your application as a Starkit, unwrap a Starkit, or show information on the contents of a Starkit. Download `sdx` from `http://www.equi4.com/pub/sk/sdx.kit` and find out more information at `http://wiki.tcl.tk/sdx`.

Note

Keep the `sdx.kit` available when working with Starkits.

To get started quickly, you can use the `sdx qwrap` command to create a Starkit from a single Tcl script. For example, create a Tcl script like the following and name the file `star1.tcl`:

```
puts "Hello from a Starkit."
```

Then run the following command to create a Starkit from this short Tcl script:

```
tclkit sdx.kit qwrap star1.tcl
```

This command creates a small file named `star1.kit`. Run the new Starkit with a command like the following:

```
   tclkit star1.kit
⇒ Hello from a Starkit.
```

Run the `sdx lsk` command to view the directory structure within the Starkit file:

```
  tclkit sdx.kit lsk star1.kit
⇒ star1.kit:
⇒                             dir  lib/
⇒         75  2008/06/19 19:25:21  main.tcl
⇒
⇒ star1.kit/lib:
⇒                             dir  app-star1/
⇒
⇒ star1.kit/lib/app-star1:
⇒         76  2008/06/19 19:25:21  pkgIndex.tcl
⇒         62  2008/06/19 19:25:21  star1.tcl
```

You can extract these files with the `sdx unwrap` command:

```
  tclkit sdx.kit unwrap star1.kit
⇒ 5 updates applied
```

This command creates a local file copy of the virtual file system in the Starkit file stored under the `star1.vfs` directory. Inside this directory, you see a file structure like the following:

```
main.tcl
lib/
lib/app-star1/
lib/app-star1/pkgIndex.tcl
lib/app-star1/star1.tcl
```

All references to the name `star1` come from the name of the original source file, `star1.tcl`. Following Starkit conventions, the package created for your application uses the prefix `app-`.
To run an expanded Starkit, use a command like the following:

```
  tclkit star1.vfs/main.tcl
⇒ Hello from a Starkit.
```

The `main.tcl` script contains the following code:

```
package require starkit
starkit::startup
package require app-star1
```

The `starkit` package provides the necessary code to initialize the Starkit virtual file system. The `starkit::startup` procedure adds the internal Starkit `lib`

directory to the Tcl `auto_path` variable. This makes all packages within the Starkit available to your application. You can add packages to the `lib` directory and then access these packages in your scripts normally. The final `package require` command `source`s your program's script.
The `pkgIndex.tcl` script defines the package index:

```
package ifneeded app-star1 1.0 \
    [list source [file join $dir star1.tcl]]
```

And the `sdx qwrap` command modifies your original Tcl script to include a `package provide` command:

```
package provide app-star1 1.0
puts "Hello from a Starkit."
```

Creating Starkits from an application consisting of multiple Tcl scripts and other extensions, encodings, image files, data files, and other auxiliary files is not much more difficult. You need to first create a directory structure similar to what the `sdx unwrap` command creates when unwrapping a Starkit. The top-level directory name is your application name followed by `.vfs`. Within it should be a file called `main.tcl`, which is executed automatically when the Starkit is run. This file should have the same content as previously shown, except the final `package require` specifies the name of your application "package." The main script of your application needs to have an appropriate `package provide` command, and you must also have a minimal `pkgIndex.tcl` file that describes how to `source` the main script. Beyond that, your script can use `source`, `package require`, and other Tcl commands to load scripts and other auxiliary files from your `.vfs` directory structure. Once you have completed development of your application, you can use the `sdx wrap` command to wrap all of the files within the `.vfs` directory into a Starkit:

```
tclkit sdx.kit wrap star1.kit
```

Note

You can also bundle platform-specific C extensions inside your Starkits. For more information on including additional files, including C-based extensions, read the documentation available at http://www.equi4.com/tclkit/.

14.7.3 Creating a Platform-Specific Executable

With Starkits, you need the separate platform-specific Tclkit interpreter to run your application. A *Starpack* is a combination of a platform-specific Tclkit interpreter and a Starkit, allowing you to provide your users with a single-file application. An advantage of this is that your users no longer need to know that you used Tcl to create the application, as the platform-specific executable hides the underlying technology from users.
To create a Starpack, use the `sdx wrap` command with the `-runtime` option:

```
  tclkit sdx.kit wrap star1 -runtime tclkit-darwin-univ-aqua
⇒ 4 updates applied
```

In this command, `star1` names the output executable. Use a name like `star1.exe` on Windows. The `-runtime` option specifies the Tclkit interpreter to use to create the new executable. You cannot create your Starpack using the Tclkit interpreter that is running the `sdx` command; of course, you can simply make a copy of the Tclkit interpreter.
When you're done, you can run your new command; for example:

```
  ./star1
⇒ Hello from a Starkit.
```

On Unix systems, you can use the `file` command to verify what you built; for example:

```
  file star1
⇒ star1: Mach-O universal binary with 2 architectures
⇒ star1 (for architecture ppc):   Mach-O executable ppc
⇒ star1 (for architecture i386):  Mach-O executable i386
```

In this case, for Mac OS X, the built executable runs on both PowerPC and Intel platforms under Mac OS X.
For more on Starkits, see http://www.equi4.com/papers/skpaper1.html. You can also download other Starkits and use the `sdx unwrap` command to see what lies inside.

15. Managing Tcl Internals

This chapter describes a collection of introspection commands that allow you to query and manipulate the internal state of the Tcl interpreter and to create helper interpreters for specialized tasks. For example, you can use these commands to see if a variable exists, to monitor all uses of a variable or command, to rename or delete a command, to handle references to undefined commands, or to create an interpreter suited to executing scripts dealing with data from untrusted sources. This chapter also discusses Tcl's features for time and date manipulation, measuring how long code execution takes, temporarily pausing script execution, and scheduling actions to take place after a delay.

15.1 Commands Presented in This Chapter

This chapter discusses the following Tcl commands for managing interpreter internals:

- `after` *`ms`*

Sleeps for *`ms`* milliseconds and then returns.

- `after` *`ms`* `?`*`script script`* `...?`

Concatenates the *`script`* arguments in the same fashion as the `concat` command, and then arranges to execute the resulting script in the global scope after *`ms`* milliseconds. Returns a unique token identifying the registered script.

- `after cancel` *`id`*

Cancels the execution of a delayed script previously registered with `after`. *`id`* is the unique identifier returned by `after` when the script was registered. If the script has already been executed, the `after cancel` command has no effect.

- `after idle` *`script`* `?`*`script`* `...?`

Concatenates the *`script`* arguments in the same fashion as the `concat` command, and then arranges to execute the resulting script in the global scope as an idle callback. The script is run exactly once, the next time the event loop is entered and there are no events to process. Returns a unique token identifying the registered script.

- `clock seconds`
  `clock microseconds`
  `clock milliseconds`

Returns the current time as an integer number of seconds, milliseconds, or microseconds since the epoch (see Section 15.3).

- `clock format` *`timeVal`* `?-`*`option value`* `...?`

Accepts a *`timeVal`* expressed as an integer number of seconds since the epoch, and returns a time and date string intended for consumption by users or external programs. See the text and the reference documentation for more details.

- `clock scan` *`inputString`* `?-`*`option value`* `...?`

Accepts a time and date string intended for users or external programs and returns a time expressed as an integer number of seconds since the epoch. Raises an error if the string cannot be converted. See the text and the reference documentation for more details.

- `clock add` *`timeVal`* `?`*`count unit`* `...? ?-`*`option value`* `...?`

Accepts a *`timeVal`* expressed as an integer number of seconds since the epoch and adds (possibly negative) *`count`* units of time to the value, returning the resulting time as an integer. See the text and the reference documentation for more details.

- `info args` *`procName`*

Returns a list whose elements are the names of the arguments to the procedure called *`procName`*, in order.

- `info body` *`procName`*

Returns the body of the procedure called *`procName`*.

- `info cmdcount`

Returns a count of the total number of Tcl commands that have been executed in this interpreter.

- `info commands ?`*`pattern`*`?`

Returns a list of all the commands that are visible in the current or specified namespace, including built-in commands, application-defined commands, procedures, ensembles, and aliases. If *`pattern`* is specified, only the command names matching *`pattern`* are returned; matching is determined using the same rules as `string match`. If the *`pattern`* includes an absolute or relative namespace reference, only the commands from the specified namespace matching the *`pattern`* are listed.

- `info complete` *`script`*

Returns `1` if *`script`* is a syntactically complete sequence of one or more Tcl commands, `0` if not. This only determines whether the commands in the script have correctly balanced braces, closed quotes, etc.; it does not ensure that the commands have the correct argument formats.

- `info default` *`procName argName varName`*

Checks to see if the argument *`argName`* to the procedure *`procName`* has a default value. If so, it stores the default value in the variable *`varName`* and returns `1`.

Otherwise, it returns `0` and puts the empty string in *`varName`*.

- `info exists` *`varName`*

Returns `1` if there exists a variable named *`varName`* in the current context, `0` if no such variable is currently accessible.

- `info globals ?`*`pattern`*`?`

Returns a list of all the global variables currently defined. If *`pattern`* is specified, only the global variable names matching *`pattern`* are returned, using `string match`'s rules for matching.

- `info hostname`

Returns the host name of the machine on which the Tcl interpreter is running.

- `info level ?`*`number`*`?`

If *`number`* is not specified, returns a number giving the current stack level (`0` corresponds to the top-level, `1` to the first level of procedure call, and so on). If *`number`* is specified, it returns a list whose elements are the name and arguments for the procedure call at level *`number`*.

- `info library`

Returns the full path name of the library directory in which standard Tcl scripts are stored.

- `info locals ?`*`pattern`*`?`

Returns a list of all the local variables defined for the current procedure, or an empty list if no procedure is active. If *`pattern`* is specified, only the local variable names matching *`pattern`* are returned, using `string match`'s rules for matching.

- `info nameofexecutable`

Returns the full name of the executable containing the Tcl interpreter.

- `info patchlevel`

Returns the current release version of the Tcl interpreter.

- `info procs ?`*`pattern`*`?`

Returns a list of the names of all procedures currently defined in the current namespace. If *`pattern`* is specified, only the procedure names matching *`pattern`* are returned, using `string match`'s rules for matching. If the *`pattern`* includes an absolute or relative namespace reference, only the procedures from the specified namespace matching the *`pattern`* are listed.

- `info script ?`*`filename`*`?`

If *`filename`* is not specified, returns the name of the script file currently being evaluated or the empty string if no script file is being executed. If *`filename`* is specified, the value to be returned by `info script` (without arguments) is set to be *`filename`* until the enclosing procedure call terminates.

- `info sharedlibextension`

Returns the platform's default file name extension for shared libraries.

- `info tclversion`

Returns the version number for the Tcl interpreter in the form *`major.minor`*.

- `info vars ?`*`pattern`*`?`

Returns a list of the names of all variables that are currently accessible. If *`pattern`* is specified, only the variable names matching *`pattern`* are returned, using `string match`'s rules for matching.

- `interp alias` *`srcPath srcCmd ?targetPath`*`? ?`*`targetCmd`*`?`
  `?`*`arg arg ...`*`?`

Creates, deletes, and manipulates command aliases between and within interpreters. If the *`targetPath`* and later arguments are omitted, it returns the current definition of the command alias with the name *`srcCmd`* in the interpreter *`srcPath`* (which may be empty to refer to the current interpreter). If *`targetPath`* is present as the empty string but no further arguments are present, the alias called *`srcCmd`* in the interpreter *`srcPath`* is deleted. If *`targetCmd`* and any optional extra arguments are present, the alias called *`srcCmd`* in the interpreter *`srcPath`* is defined to indicate the command *`targetCmd`* in the interpreter *`targetPath`* with the extra arguments all prefixed before any caller-supplied arguments.

- `interp create ?-safe? ?--? ?`*`path`*`?`

Creates the interpreter referred to by *`path`* and returns the name of the interpreter created. The optional argument `--` indicates the end of the optional arguments and should be used if *`path`* could start with a hyphen. The optional `-safe` argument results in a safe interpreter with all unsafe core Tcl commands hidden. If *`path`* is omitted, the `interp` command creates a direct child of the current interpreter with an automatically generated name.

- `interp delete` *`path`*

Deletes the interpreter referred to by *`path`* and any slave interpreters it has.

- `interp eval` *`path arg`* `?`*`arg ...`*`?`

Evaluates in the interpreter referred to by *`path`* the script formed by concatenating the given *`arg`* arguments.

- `interp expose` *`path hiddenName`* `?`*`cmdName`*`?`

Exposes the hidden command called *`hiddenName`* in the interpreter *`path`*. If *`cmdName`* is present, the command is exposed as that; otherwise it is exposed with the name it had when hidden.

- `interp hide` *`path cmdName`* `?`*`hiddenName`*`?`

Hides the command called *`cmdName`* in the interpreter *`path`*. If *`hiddenName`* is present, the command is hidden as that; otherwise it is hidden with the name it had when exposed.

- `interp invokehidden` *`path`* `?-global?` *`hiddenName`* `?`*`arg ...`*`?`

Invokes the hidden command called *`hiddenName`* in the interpreter *`path`*, passing in all following `arg` values as arguments to the hidden command itself.

- `interp limit` *`path limitType`* `?`*`option`*`? ?`*`value`*`? ...`

Configures the execution limit on the interpreter `path` of type `limitType`.

- `interp recursionlimit` *`path`* `?`*`newLimit`*`?`

Queries or sets the recursion limit (maximum depth of nested command calling) for the interpreter `path`. Omitting `newLimit` queries the current limit, and providing `newLimit` sets the limit. Note that Tcl independently imposes a hard limit based on the amount of stack space available.

- `interp share` *`srcPath channelID targetPath`*

Arranges for the channel called `channelID` in the interpreter `srcPath` to be also available with the same name in the interpreter `targetPath`.

- `interp transfer` *`srcPath channelID targetPath`*

Arranges for the channel called `channelID` in the interpreter `srcPath` to be available with the same name in the interpreter `targetPath`. The channel is not subsequently available in the interpreter `srcPath`.

- `rename` *`old new`*

Renames the command `old` to `new`, or deletes `old` if `new` is an empty string. Returns an empty string.

- `time` *`script`* `?`*`count`*`?`

Executes `script` `count` times and returns a string giving the average elapsed time per execution, in microseconds. `count` defaults to `1`.

- `trace add command` *`name ops scriptPrefix`*

Establishes a trace on the command `name` such that `scriptPrefix` is invoked whenever one of the operations given by `ops` is performed on `name`. `ops` must consist of a list of one or both of the following words: `delete`, `rename`. Returns an empty string.

- `trace add execution` *`name ops scriptPrefix`*

Establishes a trace on the command `name` such that `scriptPrefix` is invoked whenever one of the operations given by `ops` is performed during the execution of `name`. `ops` must consist of a list of one or more of the following words: `enter`, `leave`, `enterstep`, `leavestep`. Returns an empty string.

- `trace add variable` *`name ops scriptPrefix`*

Establishes a trace on the variable `name` such that `scriptPrefix` is invoked whenever one of the operations given by `ops` is performed on `name`. `ops` must consist of a list of one or more of the following words: `array`, `read`, `write`, `unset`. Returns an empty string.

- `trace info command` *`name`*
  `trace info execution` *`name`*
  `trace info variable` *`name`*

Returns a list describing each of the traces set on the command or variable *name*. Each element is itself a two-element list; the first element is the list of operations that the trace watches, and the second element is the script prefix used by the trace.

- `trace remove command` *`name ops scriptPrefix`*

  `trace remove execution` *`name ops scriptPrefix`*

  `trace remove variable` *`name ops scriptPrefix`*

If there is a trace of the indicated type set on the command or variable *`name`* with the operations and script prefix given by *`ops`* and *`scriptPrefix`*, the trace is removed, so that *`scriptPrefix`* is never again invoked. Returns the empty string. If *`name`* does not exist, an error is thrown.

- `unknown` *`cmd`* `?`*`arg arg ...`*`?`

This command is invoked by the Tcl interpreter whenever an unknown command name is encountered. *`cmd`* is the unknown command name, and the *`arg`*s are the fully substituted arguments to the command. The result returned by `unknown` is returned as the result of the unknown command.

15.2 Time Delays

The `after` command allows you to incorporate timing into your applications. It has two forms. If you invoke `after` with a single argument, the argument specifies a delay in milliseconds, and the command delays for that number of milliseconds before returning. For example,

```
after 500
```

delays for 500 milliseconds before returning. This form of the `after` command blocks, so your application cannot respond to events while the command is sleeping. In contrast, if you specify additional arguments, as in the command

```
after 5000 {puts "Time's up!"}
```

the `after` command returns immediately without any delay. However, it concatenates all of the additional arguments exactly like the `concat` command and arranges for the resulting script to be evaluated after the specified delay. The script is evaluated in the global scope as an event handler, just like the scripts for widget bindings and file events. In the previous example, a message is printed on the standard output after 5 seconds. There may be any number of `after` scripts pending at once.

The `after idle` command registers an *idle callback*:

> after idle *script* ?*script* ...?

In this form, the `script` arguments are concatenated in the same manner as with `concat`, and the resulting script is run exactly once, the next time the event loop is entered and there are no events to process. This form of the `after` command also immediately returns.
The return value of the last two forms of the `after` command is a unique identifier representing the registered script. You can pass it as an argument to the `after cancel` command to cancel the pending script if it has not yet executed.

Note

> The last forms of the `after` command require the event loop to be active to invoke the registered handlers. See Section 12.6 for more information on the event loop and event-driven programming.

As an example of using the `after` command in event-driven applications, this script uses `after` to build a general-purpose blinking utility:

```
proc blink {w option value1 value2 interval} {
    $w config $option $value1
    after $interval [list blink $w $option \
            $value2 $value1 $interval]
}
blink .b -bg red black 500
```

The `blink` procedure takes five arguments, which are the name of a widget, the name of an option for that widget, two values for that option, and a blink interval in milliseconds. The procedure arranges for the option to switch back and forth between the two values at the given blink interval. It does this by immediately setting the option to the first value and arranging for itself to be invoked again at the end of the next interval with the two option values reversed, so that option is set to the other value. The procedure reschedules itself each time it is called, so it executes periodically forever. `blink` runs “in background”: it always returns immediately, then gets re-invoked by Tk’s timer code after the next interval expires. This allows the application to do other things while the blinking occurs.

15.3 Time and Date Manipulation

Tcl's `clock` command provides a variety of time and date manipulation utilities. Many of Tcl's time-related commands express time relative to the *epoch time*, which is defined as January 1, 1970, 00:00 UTC.
The `clock seconds` command returns the current time expressed as a signed integer number of seconds since the epoch time, which is the most common representation of epoch-based time in Tcl. The `clock milliseconds` and `clock microseconds` commands return the current epoch-based time in milliseconds and microseconds respectively.

15.3.1 Generating Human-Readable Time and Date Strings

The `clock format` command accepts an integer number of seconds since the epoch time and returns a human-readable time and date string:

clock format *time* ?*-option value* ...?

`clock format` has a default format for the return value:

```
   clock format [clock seconds]
⇒ Fri Aug 29 13:05:00 PDT 2008
```

If you include a `-format` option, the argument that follows is a string that specifies how the date and time are to be formatted. The string consists of any number of characters other than the percent sign (`%`) interspersed with any number of *format groups*, which are two-character sequences beginning with the percent sign. Tcl supports a large number of format groups, many of them added in Tcl 8.5 for localization support. Table 15.1 lists the meanings of some of the more commonly used format groups. See the `clock` reference documentation for a complete list.

Table 15.1 Common `clock` Format Groups

Format group	Meaning
`%a`	Abbreviated name of the day of the week
`%A`	Full name of the day of the week
`%b` `%h`	Abbreviated name of the month
`%B`	Full name of the month
`%d`	Number of the day of the month, as two decimal digits
`%e`	Number of the day of the month, as one or two decimal digits, with a leading blank for one-digit dates
`%H`	Two-digit number giving the hour of the day (00–23) on a 24-hour clock
`%I`	Two-digit number giving the hour of the day (12–11) on a 12-hour clock
`%j`	Three-digit number giving the day of the year (001–366)
`%k`	One- or two-digit number giving the hour of the day (0–23) on a 24-hour clock
`%l`	One- or two-digit number giving the hour of the day (12–11) on a 12-hour clock
`%m`	Number of the month (01–12) with exactly two digits
`%M`	Number of the minute of the hour (00–59) with exactly two digits
`%N`	Number of the month (1–12) with one or two digits, with a leading blank for one-digit dates
`%p`	Part of the day in lower case, such as `am` or `pm`
`%P`	Part of the day in upper case, such as `AM` or `PM`
`%S`	Two-digit number of the second of the minute (00–59)
`%t`	Tab character
`%T`	Synonymous with `%H:%M:%S`
`%u`	Number of the day of the week (`1` = Monday, `7` = Sunday)
`%w`	Number of the day of the week (`0` = Sunday ; `6` = Saturday)
`%y`	Two-digit year of the century; the date is presumed to lie between 1938 and 2037 inclusive
`%Y`	Four-digit calendar year
`%z`	Current time zone, expressed in hours and minutes east (`+hhmm`) or west (`-hhmm`) of Greenwich
`%Z`	Current time zone's name
`%%`	Literal `%` character
`%+`	Synonymous with `%a %b %e %H:%M:%S %Z %Y`

Here are some examples:

```
  set time [clock seconds]
  clock format $time -format "%d %b %Y"
⇒ 29 Aug 2008
  clock format $time -format "%I:%M:%S %p"
⇒ 03:05:00 PM
```

Starting in Tcl 8.5, you can also provide a `-timezone` option to request the time zone in which the date and time are to be formatted. The time zone can be expressed in several standard formats; see the `clock` reference documentation for details.

```
  clock format $time
⇒ Fri Aug 29 15:05:00 CDT 2008
  clock format $time -timezone ":Europe/Berlin"
⇒ Fri Aug 29 22:05:00 CEST 2008
  clock format $time -timezone ":UTC"
⇒ Fri Aug 29 20:05:00 UTC 2008
```

Tcl 8.5 also introduced localization of the time and date strings. By default, `clock format` returns a time and date string localized according to the current system locale. By supplying the `-locale` option, you can specify a different locale, and `clock format` generates a localized time and date string appropriate to that locale. Many locales are supported by default; see the `clock` reference documentation for information on supplying your own localizations if they are not already provided by Tcl.

```
  clock format $time -format "%A, %B %d %Y" -locale de
⇒ Freitag, August 29 2008
  clock format $time -format "%A, %B %d %Y" -locale pt
⇒ Sexta-feira, Agosto 29 2008
  clock format $time -format "%A, %B %d %Y" -locale fr
⇒ vendredi, août 29 2008
```

15.3.2 Scanning Human-Readable Time and Date Strings

The `clock scan` command accepts a human-readable time and date string and returns an integer value representing the number of seconds since the epoch:

clock format *inputString* ?*-option value* ...?

For compatibility with versions of Tcl prior to 8.5, the `clock scan` command applies some heuristics to attempt to parse the string without guidance; for example:

```
  clock format [clock scan "3:37 pm"]
⇒ Fri Oct 31 15:37:00 CDT 2008
  clock format [clock scan "August 29, 1965"]
⇒ Sun Aug 29 00:00:00 CDT 1965
  clock format [clock scan "4:15 pm February 8, 2037"]
⇒ Sun Feb 08 16:15:00 CST 2037
```

However, to reduce ambiguity in Tcl 8.5 and later, you should provide the `-format` option to `clock scan` followed by a string describing the expected format of the input. The string can consist of any number of characters other than the percent sign (`%`), interspersed with any number of format groups. The format groups are the same as supported for the `clock format` command described in the previous section. `clock scan` raises an error if the string does not match the specified format:

```
  clock format [clock scan "Sept 9, 1961" -format "%B %e, %Y"]
⇒ Sat Sep 09 00:00:00 CDT 1961
  clock format [clock scan "20080829T142305" \
      -format "%Y%m%dT%H%M%S"]
⇒ Fri Aug 29 14:23:05 CDT 2008
  clock scan "1:23am" -format "%B"
∅ input string does not match supplied format
```

Note

In general, the `clock scan` format group interpretation is more generous than that of `clock format`. For example, `%d` results in a two-digit day of the month, with a leading `0` if necessary, with `clock format`; in contrast, it matches a one- or two-digit day, with optional leading space, with `clock scan`. Similarly, `%B` produces the full month name with `clock format`, whereas `clock scan` allows the month name in abbreviated or full form, or any unique prefix of either form.

The `clock scan` command in Tcl 8.5 and later also supports localized time and date representations. By default, the time and date string is interpreted according to the system locale. However, you can use the `-locale` option in combination with `-format` to specify an alternate locale for parsing the string:

```
  clock scan "Sexta-feira, Agosto 29 2008" \
      -format "%A, %B %d %Y"
∅ input string does not match supplied format
  clock scan "Sexta-feira, Agosto 29 2008" \
      -format "%A, %B %d %Y" -locale pt
⇒ 1219986000
  clock scan "vendredi, août 29 2008" \
      -format "%A, %B %d %Y" -locale fr
⇒ 1219986000
```

15.3.3 Performing Clock Arithmetic

For compatibility with versions of Tcl prior to 8.5, the `clock scan` command supports simple clock arithmetic. In addition to a small set of keywords such as `now`, `today`, `yesterday`, and `tomorrow`, you can specify any combination of units of time to add or subtract; for example:

```
  clock format [clock scan "now"]
⇒ Tue Jan 27 02:49:10 CST 2009
  clock format [clock scan "12:15pm yesterday"]
⇒ Mon Jan 26 12:15:00 CST 2009
  clock format [clock scan "tomorrow - 2 months + 3 weeks"]
⇒ Fri Dec 19 00:00:00 CST 2008
```

Tcl 8.5 introduced the `clock add` command to eliminate the ambiguity of the `clock scan` command in clock arithmetic:

clock add *timeVal* ?*count unit* ...? ?-*option value* ...?

The first argument to `clock add` is an integer number of seconds relative to the epoch time. The remaining arguments are integers and keywords in alternation, where the keywords are chosen from `seconds`, `minutes`, `hours`, `days`, `weeks`, `months`, or `years` or any unique prefix of such a word. The integers specify the number of units to add to—or subtract from, in the case of a negative integer—the base time:

```
  set time [clock scan "2008-01-30 05:00:00" \
      -format "%Y-%m-%d %H:%M:%S"]
  clock format $time
⇒ Wed Jan 30 05:00:00 CST 2008
  clock format [clock add $time 5 weeks]
⇒ Wed Mar 05 05:00:00 CST 2008
  clock format [clock add $time 1 month]
⇒ Fri Feb 29 05:00:00 CST 2008
  set time [clock scan {2004-10-30 05:00:00} \
      -format {%Y-%m-%d %H:%M:%S}]
  clock format $time
⇒ Sat Oct 30 05:00:00 CDT 2004
  clock format [clock add $time 1 day]
⇒ Sun Oct 31 05:00:00 CST 2004
  clock format [clock add $time 24 hours]
⇒ Sun Oct 31 04:00:00 CST 2004
```

Note

The `clock add` command attempts to handle gracefully situations such as the start or end of Daylight Savings Time and the change from Julian to Gregorian calendars. See the `clock` reference documentation for complete information.

15.4 Timing Command Execution

The `time` command is used to measure the performance of Tcl scripts. It takes two arguments, a script and an optional repetition count:

```
    time {format "%d%s%f" 123 xyz 4.56} 100000
⇒ 3.85866 microseconds per iteration
```

`time` executes the given script the number of times given by the repetition count (which defaults to `1`), divides the total elapsed time by the repetition count, and returns a message similar to the preceding example, giving the average number of microseconds per iteration. The repetition count allows reasonably accurate measurements even when the clock resolution is less than ideal.

Note

In general, to make accurate timing measurements, experiment with the repetition count until the total time for the `time` command to run is a second or so.

15.5 The `info` Command

The `info` command provides information about the state of the Tcl interpreter through introspection. It has a rich set of options, which are discussed in the following sections.

15.5.1 Information about Variables

Several of the `info` options provide information about variables. `info exists` returns a `1` or `0` value, indicating whether or not there exists a variable (including an array element) with a given name:

```
  set x 24
  info exists x
⇒ 1
  unset x
  info exists x
⇒ 0
  info exists env(PATH)
⇒ 1
```

The options `vars`, `globals`, and `locals` return lists of variable names that meet certain criteria. `info vars` returns the names of all variables accessible at the current level of procedure call; `info globals` returns the names of all global variables, regardless of whether or not they are accessible; and `info locals` returns the names of local variables, including arguments to the current procedure, if any, but not global variables. In each of these commands, an additional pattern argument may be supplied. If the pattern is supplied, only variable names matching that pattern (using the rules of `string match`) are returned.
For example, suppose that the global variables `global1` and `global2` have been defined and that the following procedure is being executed:

```
proc test {arg1 arg2} {
    global global1
    set local1 1
    set local2 2
   ...
}
```

Executing the following commands within the procedure would produce the following results:

```
  info vars
⇒ global1 arg1 arg2 local2 local1
  info globals
⇒ global2 global1
  info locals
⇒ arg1 arg2 local2 local1
  info vars *al*
⇒ global1 local2 local1
```

In addition, the pattern argument to `info vars` may be preceded by a namespace to list the variables defined in that namespace:

```
    namespace eval example {
        variable var1
        variable var2
        variable anotherVar
    }
    info vars example::var*
⇒ ::example::var2 ::example::var1
```

15.5.2 Information about Procedures

Another group of `info` options provides information about procedures. The command `info procs` returns a list of all the Tcl procedures that are defined in the current namespace. Like `info vars`, it takes an optional pattern argument that restricts the names returned to those that match a given pattern or selects another namespace to examine. `info body`, `info args`, and `info default` return information about the definition of a procedure:

```
    proc maybePrint {a b {c 24}} {
        if {$a < $b} {
            puts "c is $c"
        }
    }
    info body maybePrint
⇒
        if {$a < $b} {
            puts "c is $c"
        }

    info args maybePrint
⇒ a b c
    info default maybePrint a x
⇒ 0
    info default maybePrint c x
⇒ 1
    set x
⇒ 24
```

`info body` returns the procedure's body exactly as it was specified to the `proc` command. `info args` returns a list of the procedure's argument names, in the same order in which they were specified to `proc`. `info default` returns information about an argument's default value. It takes three arguments: the name of a procedure, the name of an argument to that procedure, and the

name of a variable. If the given argument has no default value (for example, `a` in the preceding example), `info default` returns `0`. If the argument has a default value (`c` in the preceding example), `info default` returns `1` and sets the variable to hold the default value for the argument.
As an example of how you might use the commands from the previous paragraph, here is a Tcl procedure that writes a Tcl script file. The script contains Tcl code in the form of `proc` commands that re-create all of the procedures in the global namespace of the interpreter. The file can then be `source`d in some other interpreter to duplicate the procedure state of the original interpreter. The procedure takes a single argument, which is the name of the file to write:

```
proc printProcs {file} {
    set f [open $file w]
    foreach proc [info procs] {
       set argList {}
       foreach arg [info args $proc] {
           if {[info default $proc $arg default]} {
               lappend argList [list $arg $default]
           } else {
               lappend argList [list $arg]
           }
       }
       puts $f [list proc $proc $argList \
               [info body $proc]]
    }
    close $f
}
```

`info` provides one other option related to procedures: `info level`. If `info level` is invoked with no additional arguments, it returns the current procedure invocation level: `0` if no procedure is currently active, `1` if the current procedure was called from the top level, and so on. If `info level` is given an additional argument, the argument indicates a procedure level, and `info level` returns a list whose elements are the name and actual arguments for the procedure at that level. For example, the following procedure prints out the current call stack, showing the name and arguments for each active procedure:

```
proc printStack {} {
    set level [info level]
    for {set i 1} {$i < $level} {incr i} {
        puts "Level $i: [info level $i]"
    }
}
```

15.5.3 Information about Commands

The `info commands` subcommand is similar to `info procs` except that it returns information about all currently visible commands, not just procedures. If invoked with no arguments, it returns a list of the names of all commands visible in the current namespace context. If an argument is provided, it is a pattern in the sense of `string match` with an optional namespace prefix (just as with `info procs` and `info vars`), and only command names matching that pattern (in the given namespace if the prefix is there) will be returned.

```
  info commands for*
⇒ for format foreach
  namespace eval example {
      proc foo {} {return foobar}
      proc bar {} {return barfoo}
  }
  info commands example::*
⇒ ::example::foo ::example::bar
```

The command `info cmdcount` returns a number indicating how many commands have been executed in this Tcl interpreter. It may be useful during performance tuning to see how many Tcl commands are being executed to carry out various functions, and it is also a value that may be controlled using an interpreter resource limit (as discussed in Section 15.11.4).
The command `info script` indicates whether a script file is currently being processed. If so, the command returns the name of the innermost nested script file that is active. If there is no active script file, `info script` returns an empty string. This command is used mostly for working out the location of files that are placed in the same directory as the script file, like this:

```
set otherFile [file dirname [info script]]/other.tcl
```

Note

You can also override the current script file name up until the end of the currently executing procedure by passing the new script file name as an argument to `info script`, which is used when writing customized replacements for the `source` command. This is the way to perform preprocessing of a script before evaluation, but it is only rarely useful.

The command `info complete` is used when parsing input from a channel (such as `stdin`) to determine whether a string represents a syntactically complete sequence of Tcl commands. This is used mainly when writing a handler that allows the execution of interactive scripts as part of the processing of a program; when `tclsh` is executing without a script argument, its main read-evaluate loop uses `info complete` to determine when to pass a string to the Tcl interpreter for evaluation.

15.5.4 The Tcl Interpreter Version and Other Runtime Environment Information

The `info tclversion` command returns the version number for the Tcl interpreter in the form *`major.minor`*, such as `8.5`. Both *`major`* and *`minor`* are decimal strings. If a new release of Tcl contains only backward-compatible changes such as bug fixes and new features, its minor version number increments and the major version number stays the same. If a new release contains changes that are not backward-compatible, so that existing Tcl scripts or C code that invokes Tcl's library procedures will have to be modified, the major version number increments and the minor version number resets to 0. The command `info patchlevel` returns more detailed version information that indicates the exact patch level of the Tcl interpreter, such as `8.5.3`; this is usually useful only when submitting bug reports or determining whether to upgrade.

The command `info library` returns the full path name of the Tcl library directory. This directory holds standard scripts used by Tcl, such as a default definition for the `unknown` procedure described in Section 15.10.

The command `info hostname` returns the host name of the machine on which the Tcl interpreter is running. Note that this command returns only the primary name of the machine. Some machines (especially server systems with multiple network connections) may have multiple names, and a more sophisticated approach to obtaining the name may be needed.

The commands `info nameofexecutable` and `info sharedlibextension` respectively return the name of the executable that was used to invoke the Tcl interpreter and the standard shared library file name extension used on the platform on which the Tcl interpreter is running. `info nameofexecutable` is useful in situations where you want to invoke another Tcl interpreter as a separate process via `exec` or `open`, and `info sharedlibextension` is useful when creating cross-platform Tcl scripts that use `load` to access extension modules written in programming languages like C.

15.6 Tracing Operations on Simple Variables

The `trace` command allows you to monitor the usage of a number of aspects of the Tcl interpreter, including variables. Such monitoring is called *tracing*. If a trace has been established on a variable, a Tcl command is invoked whenever the variable is read, written, or unset. Traces can be used for a variety of purposes, such as

- Monitoring the variable's usage (for example, by printing a message for each read or write operation)
- Propagating changes in the variable to other parts of the system (for example, to ensure that a particular widget always displays the picture of a person named in a given variable)
- Restricting usage of the variable by rejecting certain operations (for example, generating an error on any attempt to change the variable's value to anything other than a decimal string) or by overriding certain operations (for example, re-creating the variable whenever it is unset)

Here is a simple example that prints a message when either of two variables is modified:

```
trace add variable color write pvar
proc pvar {name element op} {
    upvar 1 $name x
    puts "Variable $name set to $x"
}
```

In this example, the `trace` command arranges to invoke the procedure `pvar` whenever the variable `color` is written. The argument `variable` specifies that a variable trace is being created, `color` gives the name of the variable, `write` specifies a set of operations to trace (a list of any combination of `read`, `write`, and `unset`), and the last argument is a command to invoke.
Whenever `color` is modified, Tcl invokes `pvar` with three additional arguments appended: the variable's name; the variable's element name if it is an array element, or an empty string otherwise (the `pvar` procedure ignores this argument); and an argument indicating what operation was actually invoked. The operation argument is `read` when the variable is read, `write` when it is

written, and `unset` when the variable is deleted. For example, if the command `set color purple` is executed, Tcl evaluates the command `pvar color {} write` because of the trace.
The `pvar` procedure does two things. First, the procedure uses `upvar` to make the variable's value accessible inside the procedure as local variable `x`. Then it prints out the variable's name and value on standard output. For the access in the preceding paragraph the following message would be printed:

```
Variable color set to purple
```

Write traces are invoked after the variable's value has been modified but before returning the new value as the result of the write. The `trace` command can write a new value into the variable to override the value specified in the original write, and this value is returned as the result of the traced write operation. Read traces are invoked just before the variable's result is read. The `trace` command can modify the variable to affect the result returned by the read operation. Tracing is temporarily disabled for a variable during the execution of read and write `trace` commands. This means that a `trace` command can access the variable without causing traces to be invoked recursively.
If a read or write trace returns an error of any sort, the traced operation is aborted. This can be used to implement read-only variables, for example. Here is a script that forces a variable to have a positive integer value and rejects any attempts to set the variable to a non-integer value:

```
trace add variable size write forceInt
proc forceInt {name element op} {
    upvar 1 $name x ${name}_old x_old
    if {![regexp {^\d+$} $x]} {
        set x $x_old
        error "value must be a positive integer"
    }
    set x_old $x
}
```

By the time the `trace` command is invoked, the variable has already been modified, so if `forceInt` wants to reject a write, it must restore the old value of the variable. To do this, it keeps a shadow variable with the suffix `_old` to hold the previous value of the variable. If an illegal value is stored in the variable, `forceInt` restores the variable to its old value and generates an error:

```
  set size 47
⇒ 47
  set size red
∅ can't set "size": value must be a positive integer
  set size
⇒ 47
```

You can use traces to create a read-only variable. The simplest way to implement this is to store the value to which the variable is to be reset as part of the trace callback itself; this is possible because the trace callback is really a script fragment and not just a command name. The additional argument provided when registering the trace becomes an additional argument to the callback procedure; for example:

```
  set pi [expr {4*atan(1)}]
  trace add variable pi write [list constant $pi]
  proc constant {value name element op} {
      upvar 1 $name var
      set var $value
      error "variable is a constant"
  }
  set pi 3
∅ can't set "pi": variable is a constant
```

Note

When generating script fragments for callbacks and other deferred evaluations, it is highly advisable to use the `list` command to do the generation as was just demonstrated. This applies the correct amount of quoting to ensure that the value is relayed to the callback procedure exactly as it was provided at the creation of the callback, without interference from the Tcl interpreter.

It is legal to set a trace on a nonexistent variable; the variable continues to appear to be unset even though the trace exists. For example, you can set a read trace on an array, and then use it to create new array elements automatically the first time they are read. Unsetting a variable removes the variable and any traces associated with the variable, then invokes any unset traces for the variable. It is legal, and not unusual, for an unset trace to immediately reestablish itself on the same variable so that it can monitor the variable if it should be re-created in the future.

To delete a trace, invoke `trace remove variable` with the same arguments passed to `trace add variable`. For example, the trace on `color` in the earlier example can be deleted with the following command:

```
trace remove variable color write pvar
```

If the arguments to `trace remove variable` do not match the information for any existing trace exactly, the command has no effect.
The command `trace info variable` returns information about the traces currently set for a variable. It is invoked with an argument consisting of a variable name, as in the following example:

```
    trace info variable color
⇒ {write pvar}
```

The return value of `trace info variable` is a list, each of whose elements describes one trace on the variable. Each element is itself a list with two elements, which give the list of operations traces and the command for each trace. The traces appear in the result list in the order in which they are invoked. If the variable specified to `trace info variable` is an element of an array, only traces on that element are returned; traces on the array as a whole are not returned.

Note

It is possible to use traces to create very unclear code, such as making the whole world change when you read or write a variable. This is considered very poor style. If an operation may have significant side effects, it should (usually) be implemented as a command invocation.

15.7 Tracing Array Variables

With `trace` you can trace not only normal variables, but also array elements and whole arrays. Tracing an array element is exactly the same as tracing a simple variable, except that the variable name is split into two parts (the array name and the element name) when passed to the trace callback command; for example:

```
  set a(length) 42
  trace add variable a(length) write pAry
  proc pAry {aName eName op} {
      upvar 1 $aName ary
      puts [format "Notice: %s(%s) has changed to %s" \
              $aName $eName $ary($eName)]
  }
  incr a(length)
⇒ Notice: a(length) has changed to 43
  43
```

When tracing whole arrays, there are finer nuances. First, the read, write, and unset traces fire when any element of the array is accessed (according to the type of access), *except* for the case where the `upvar` command was used to create a link to an element of the traced array. Second, an additional type of trace is available to handle accesses that need to know the whole set of elements, such as when processing the `array get` or `array names` commands: the `array` trace type. This fires whenever the array is accessed as a whole; when it triggers, it never passes the name of an element.

```
  set a(b) c
  trace add variable a {read write array} pWholeAry
  proc pWholeAry {aName eName op} {
      puts "Traced: $op on ${aName}($eName)"
  }
  append a(b) c
⇒ Traced: write on a(b)
⇒ cc
  array set a {c d e f}
⇒ Traced: array on a()
⇒ Traced: write on a(c)
⇒ Traced: write on a(e)
  array get a {[a-c]}
⇒ Traced: array on a()
⇒ Traced: read on a(b)
⇒ Traced: read on a(c)
⇒ b cc c d
```

Note

Array traces are used to implement the global `env` array. Those traces cause the array to be rebuilt every time it is written to or read from. This means that you are *strongly* advised to avoid using `upvar` to link a local variable to any element of the `env` array; always use `global` to bring the whole array into scope. This re-creation behavior is also why accesses to `env` are slow; it is always more efficient to use some other mechanism when you do not actually need access to the environment variables.

15.8 Renaming and Deleting Commands

The `rename` command is used to change the command structure of an application. It takes two arguments:

rename *old new*

`rename` does just what its name implies: it renames the command that used to have the name *`old`* so that it now has the name *`new`*. It is an error if *`new`* already exists as a command when you invoke `rename`.
You can also use `rename` to delete a command by invoking it with an empty string as the *`new`* name. For example, the following script removes event loop processing from an interpreter by deleting the relevant commands:

```
foreach cmd {after fileevent update vwait} {
    rename $cmd {}
}
```

Any Tcl command may be renamed or deleted, including the built-in commands as well as procedures and commands defined by an application. Take care when renaming or deleting a built-in command, since it will break scripts that depend on the command (possibly including some other built-in commands), but in some situations it can be useful. For example, the `exit` command as defined by Tcl exits the process immediately. If an application wants to have a chance to clean up its internal state before exiting, it can create a “wrapper” around `exit` by redefining it:

```
rename exit exit.old
proc exit {{status 0}} {
    # application-specific cleanup
    # ...
    exit.old $status
}
```

In this example, the `exit` command is renamed `exit.old` and a new `exit` procedure is defined, which performs the cleanup required by the application and then calls the renamed command to exit the process. This allows existing scripts that call `exit` to be used without change, while still giving the application an opportunity to clean up its state.

15.9 Tracing Commands

The `trace` command may be used to trace command-related events as well as variable-related events. By using `trace add command`, you can track changes to a command's name (using the `rename` operation) and detect when the command is deleted (the `delete` operation). This is particularly useful in situations where the command has associated with it some resources that should be deleted at the same time as the command. For example, the following procedure implements a simple logging system that closes the log file when the logging command is deleted:

```
proc log {name filename} {
    set f [open $filename a]
    interp alias {} $name {} logCommand $f
    trace add command $name delete [list logDone $f]
    return $name
}
proc logCommand {f args} {
    set now [clock seconds]
    set time [clock format $now -format "%D %T: "]
    puts $f $time[join $args " "]
}
proc logDone {f old new op} {
    puts "Renaming \"$old\" to \"$new\" ($op)"
    close $f
}
```

In this example, the `log` command creates a new command that logs to the given file by creating an interpreter alias (see Section 15.11.1 for more details) to the `logCommand` command that implements the logging command. To ensure that the file is closed in a timely fashion, the `log` command then adds a delete trace that ensures that the logging channel is disposed of when the command to write to it is destroyed. The command trace callback, implemented by `logDone`, is invoked whenever the command is either deleted or renamed, as selected in the `trace add command` invocation. The callback is invoked with three extra arguments: what the command was renamed from, what it is being renamed to (or the empty string in the case of deletion), and what operation is being performed. The following illustrates the command in use:

```
  log example eg.log
⇒ example
  example This is a message
  example This is another message
  rename example eg
  eg This is a third example
  rename example {}
⇒ Renaming "eg" to "" (delete)
  read [open eg.log]
⇒ 05/07/05 14:56:14: This is a message
  05/07/05 14:56:19: This is another message
  05/07/05 14:57:02: This is a third example
```

This example also illustrates another important technique for working with traces: adding user-defined arguments to the trace callback. These additional arguments always come before the arguments added by the trace callback process. This is how `logDone` discovers what channel to close.

Commands may also have their execution behavior traced using `trace add execution`, which is particularly useful for debugging complex scripts. Four kinds of execution events may be traced; to receive notification of when a command is invoked, you should put an `enter` trace on it, and to find out what the command's result is, you should put a `leave` trace on it. Finer-grained information about the execution of a command is also available. With the `enterstep` and `leavestep` execution traces it is possible to find out about all the command entry and completion behavior that happens during the execution of a command (this is most obvious during the processing of a procedure, but it can be applied to any other Tcl command as well).

With `enter` and `enterstep` traces, two arguments are added to the callback command. The first is the list of fully substituted arguments to the command, and the second is the operation that triggered the trace callback. With `leave` and `leavestep` traces, four arguments are appended; as with `enter` traces, the first argument is the list of fully substituted arguments, and the last argument is the operation name. However, the second argument is the return code just as would be seen by `catch` at that point (`0` for success, `1` for error, etc.), and the third argument is the result of the command.

```
    proc tracer {args} {puts TRACE:$args}
    proc subcmd {s} {string length $s}
    proc demo {args} {
        expr {[llength $args]+[subcmd $args]}
    }
    trace add execution demo {enter leave} tracer
    demo a b c
⇒   TRACE:{demo a b c} enter
    TRACE:{demo a b c} 0 8 leave
    8
    demo "another demo"
⇒   TRACE:{demo {another demo}} enter
    TRACE:{demo {another demo}} 0 15 leave
    15
    trace remove execution demo {enter leave} tracer
    trace add execution demo {enterstep leavestep} tracer
    demo a b c
⇒   TRACE:{expr {[llength $args]+[subcmd $args]}} enterstep
    TRACE:{llength {a b c}} enterstep
    TRACE:{llength {a b c}} 0 3 leavestep
    TRACE:{subcmd {a b c}} enterstep
    TRACE:{string length {a b c}} enterstep
    TRACE:{string length {a b c}} 0 5 leavestep
    TRACE:{subcmd {a b c}} 0 5 leavestep
    TRACE:{expr {[llength $args]+[subcmd $args]}} 0 8 leavestep
    8
```

Like variable traces, `trace info` is used to discover information about command and execution traces, and `trace remove` can be used to delete them.

```
    trace info execution demo
⇒   {{enterstep leavestep} tracer}
```

15.10 Unknown Commands

The Tcl interpreter provides a special mechanism for dealing with unknown commands. If the interpreter discovers that the command name specified in a Tcl command does not exist, it checks for the existence of a command named `unknown`. If there is such a command, the interpreter invokes `unknown` instead of the original command, passing the name and arguments for the nonexistent command to `unknown`. For example, suppose that you type the following commands:

```
set x 24
```

```
createDatabase library $x
```

If there is no command called `createDatabase`, the following command is invoked:

```
unknown createDatabase library 24
```

Notice that substitutions are performed on the arguments to the original command before `unknown` is invoked. Each argument to `unknown` consists of one fully substituted word from the original command.
The `unknown` procedure can do anything it likes to carry out the actions of the command, and whatever it returns is returned as the result of the original command. For example, the following procedure checks to see if the command name is an unambiguous abbreviation for an existing command; if so, it invokes the corresponding command:

```
proc unknown {name args} {
    set cmds [info commands $name*]
    if {[llength $cmds] != 1} {
        error "unknown command \"$name\""
    }
    uplevel 1 $cmds $args
}
```

Note that when the command is re-invoked with an expanded name, it must be invoked using `uplevel` so that the command executes in the same variable context as the original command.
The Tcl script library includes a default version of `unknown` that performs the following functions, in order:

1. If the command is a procedure that is defined in a library file, `source` the file to define the procedure, then re-invoke the command. This is called *autoloading*, described in Chapter 14.
2. If a program exists with the name of the command, use the `exec` command to invoke the program. This feature is called *autoexec*. For example, on a Unix system you can type `ls` as a command, and `unknown` will invoke `exec ls` to list the contents of the current directory. If the command does not specify redirection, autoexec arranges for the command's standard input, standard output, and standard error to be redirected to the corresponding channels of the Tcl application. This is different from the normal behavior of `exec`, but it allows interactive programs such as `more` and `vi` to be invoked directly from a Tcl application.

3. If the command name has one of several special forms such as `!!`, compute a new command using history substitution and invoke it. If, for example, the command is `!!`, the previous command is invoked again. See Chapter 16 for more information on history substitution.
4. If the command name is a unique abbreviation for an existing command, the abbreviated command name is expanded and the command is invoked again.

Note

The last three actions are intended as conveniences for interactive use, and they occur only if the command was invoked interactively. You should not depend on these features when writing scripts. For example, you should never use autoexec in scripts; always use the `exec` command explicitly.

If you do not like the default behavior of the `unknown` procedure, you can write your own version or modify the library version to provide additional functions; the apparent nature of the Tcl language can be greatly changed through this. If you do not want any special actions to be taken for unknown commands, you can just delete the `unknown` procedure, in which case errors occur whenever unknown commands are invoked.

Note

The one feature of the autoexec system that is generally useful is the command `auto_execok`, which takes the name of an external command that you might want to run with `exec` and returns a list of words to use with `exec` that describe how to actually do it. This includes searching the `PATH` for the program, adding suitable code for invoking the shell if it was a shell internal command rather than a program, and so on. For example, on Windows the `start` command is actually a shell internal, but it is extremely useful for opening many different types of files, just as in Windows Explorer. So you could use it like this to open a saved web page:

```
exec {*}[auto_execok start] example.html
```

15.11 Slave Interpreters

Slave interpreters are Tcl interpreters that are children of some other interpreter (called the master). They are used in cases where there is a need to evaluate a Tcl script in a context that is substantially different from that of the main Tcl program, such as

- Processing untrusted code
- Handling a plugin mechanism for a larger program that does not expose the details of the main body of the application
- Creating a friendlier API for configuring an application
- Isolating executions of some code from each other
- Limiting the resources that may be used for a particular operation

Slave interpreters' variables are independent of the variables of the master interpreter or any other slave interpreters, with the notable exception of the shared global `env` array. This makes it easier to keep different spaces of variables apart.

For example, you could have a configuration file format for your program without any danger that temporary variables used in the configuration file will interfere with the operation of the main program. Or you could have a chess game system where the state of the board and the rules of the game are enforced by the master interpreter, but each player's code is in an independent slave interpreter, making it easy to show that the players are not interfering with each other.

Slave interpreters are manipulated using the `interp` command. It is possible to nest slave interpreters within other slave interpreters, and the subcommands of the `interp` command indicate what interpreter they are manipulating by giving a path as a list of interpreter names, starting at the current interpreter. The slave interpreters are all completely independent from each other except for the global `env` array, which is normally shared.

A slave interpreter is created (in the current thread if threading is enabled) using the `interp create` command, which takes an optional interpreter name. Scripts may then be evaluated in this interpreter using `interp eval`, and the interpreter can then be deleted with `interp delete`.

Note

Using `interp eval` does not start a separate thread of execution. You can have multiple interpreters per thread, just as you can have multiple threads per OS process. Interpreters provide script-level isolation and different execution and global variable contexts. Parallel execution can be added through the use of multiple threads, which requires a threaded build of Tcl and the use of the `Thread` extension. Full isolation requires the use of multiple processes and interprocess communication, with all the overhead that implies.

As you can see from the following example, a slave interpreter makes creating an application-specific language easy:

```
  set slave [interp create]
⇒ interp0
  interp eval $slave rename expr  calculate
  interp eval $slave rename value set
  interp eval $slave rename incr  advance
  interp eval $slave {
      value x 3
      calculate {1 + 2 + $x + [advance x] + 5}
  }
⇒ 15
  interp delete $slave
```

When you create a slave interpreter, a command with the same name is also created. This allows you to invoke many operations directly, and to delete the interpreter by deleting the command. This means that the previous example could have been written like this:

```
  set slave [interp create]
⇒ interp0
  $slave eval rename expr  calculate
  $slave eval rename value set
  $slave eval rename incr  advance
  $slave eval {
      value x 3
      calculate {1 + 2 + $x + [advance x] + 5}
  }
⇒ 15
  rename $slave {}
```

15.11.1 Command Aliases

Sometimes it is necessary for scripts running in slave interpreters to be able to trigger the running of commands in their masters. For example, a network server configuration script will want to tell the server how to configure itself, even if it does not have the power to directly run the configuration code itself. This powerful feature is enabled by setting up an alias command in the slave that triggers the running of a command in the master using `interp alias`. You can get the list of aliases defined on an interpreter by using the `interp aliases` command, and individual alias configurations can be obtained by using `interp alias` with fewer arguments than would be necessary to define an alias.

```
  interp create configurator
  interp alias configurator service {} initNamedService
  interp alias configurator usePort {} setServicePort
  interp alias configurator dump    {} parray config
  proc initNamedService {name} {
      global servName
      set servName $name
  }
  proc setServicePort {port} {
      global servName config
      if {![string is integer -strict $port]} {
          return -code error \
                  "\"$port\" is not an integer"
      }
      set config($servName) $port
  }
  interp eval configurator {
      service http
      usePort 80
      service "http-alt"
      usePort 8080
      dump
  }
⇒ config(http)     = 80
  config(http-alt) = 8080
  interp eval configurator {
      service {Whale watching!}
      usePort {Key West}
  }
∅ "Key West" is not an integer
```

As a convenience, the per-slave command created when you create the slave interpreter has an `alias` subcommand, though this allows only the definition of aliases from the slave interpreter into the master. (The full `interp alias` command has a wider repertoire, including allowing aliases to be defined from one slave interpreter to another.) This means that the definition of the `service`, `usePort`, and `dump` aliases from the previous example could have been done like this:

```
configurator alias service initNamedService
configurator alias usePort setServicePort
configurator alias dump    parray config
```

Aliases can even be defined within an interpreter by using an empty string for the interpreter name, which is useful for defining one command in terms of another:

```
interp alias {} print      {} puts stdout
interp alias {} printError {} puts stderr
print      "This message goes to standard output"
printError "and this one goes to standard error!"
```

15.11.2 Safe Slave Interpreters and Hidden Commands

One of the most important uses for slave interpreters is the parsing of Tcl scripts that are not trusted by the master interpreter. Examples of this include the parsing of Tcl code that is embedded within a web page, e-mail message, or some other context where the creator of the code is not necessarily known. Tcl provides a mechanism for handling this situation called a *safe interpreter*. Safe interpreters are slave interpreters that have a special flag set at creation time and that hide all their unsafe commands (for example, `exec`, `open`, `socket`, `source`) so that they may not be called unless explicitly authorized to do so by the master interpreter. This mechanism, analogous to the way that an operating system handles the security of processes, is a very strong mechanism because there is no way to perform any state-changing operation within the Tcl language without executing a command. Although a safe interpreter can change its internal state, such as by creating procedures or setting variables, by default no commands are exposed in the safe interpreter for interacting with its master or the system executing the application. And unlike normal slave interpreters, safe slaves do not have access to the `env` variable and do not have a predefined `unknown` command (described in Section 15.10).
You create safe interpreters by passing the `-safe` flag to `interp create`. You then use them exactly like any other slave interpreter, though the default set of commands is somewhat more restricted:

```
  set safe [interp create -safe]
⇒ interp0
  interp eval $safe {
      file delete -force /home
  }
∅ invalid command name "file"
```

A master interpreter should expose more dangerous behavior only in a controlled fashion to the safe interpreter using aliases. This allows access to specific parts of the behavior of a command without opening up access to all parts of it:

```
  interp alias $safe file {} safeFile
  proc safeFile {subcommand args} {
     switch -- $subcommand {
        dirname - extension - rootname - tail {
           return [file $subcommand {*}$args]
        }
        default {
           return "unknown subcommand \"$subcommand\""
        }
     }
  }
  interp eval $i {
     file extension example.tcl
  }
⇒ .tcl
  interp eval $i {
     file delete -force /home
  }
∅ unknown subcommand "delete"
```

Another common example of selectively exposing access to prohibited commands within a safe interpreter is controlled access to the `source` command. You very well might want to `source` files or load packages from a safe interpreter, but by default the `source` command is not exposed in a safe interpreter.

When the unsafe functionality is removed from a safe interpreter during its creation, the unsafe commands are not actually deleted. Instead, the interpreter hides them. The master interpreter may invoke hidden commands using the `interp invokehidden` command, as can be seen in the following example. This implements a safer version of the `source` command that checks to see whether the file that is to be `source`d is really a Tcl script file in the Tcl library, otherwise refusing and claiming that the file does not exist:

```
interp alias $i source {} safeSource $i
proc safeSource {slave filename} {
    # Get rid of elements like .. from the name
    # Very important in real code!
    set fullFilename [file normalize $filename]

    # Get the parts of the filename to examine
    set dir [file dirname $fullFilename]
    set ext [file extension $fullFilename]

    # Perform the safety check
    if {[string first [info library] $dir] != 0 ||
            $ext ne ".tcl"} {
        return -code error \
                "unknown file \"$filename\""
    }

    # Really source the file
    return [interp invokehidden $slave \
            source $fullFilename]
}
```

The master interpreter may use the `interp hide` and `interp expose` commands to manipulate the set of commands hidden and exposed by a slave interpreter, modifying the profile of code that may run successfully within that safe interpreter. In the following example the `pwd` command has been exposed so that code executing in the safe slave can discover its current directory, and the `vwait` and `update` commands have been hidden so the event loop cannot be entered:

```
# Expose [pwd] to make a low-priority information leak
interp expose $i pwd
# Hide the ways to run the event loop
interp hide $i update
interp hide $i vwait
```

Note

While safe interpreters may create their own slave interpreters by default, all the interpreters that they create are forced to be safe interpreters, and many of the subcommands of the `interp` command are not available. Many extensions to Tcl also restrict themselves in safe interpreters. For example, Tk permits the use of only some of its widgets from within a safe interpreter, blocking access to anything that could be used to obscure the enclosing application, and it also prevents the script from having access to things like system-wide configuration options or the clipboard.

15.11.3 Transferring Channels between Interpreters

By default, slave interpreters do not have access to their parents' I/O channels. If an interpreter wishes to give access to a channel to one of its slaves, it should use `interp share` or `interp transfer`. `interp share` allows a second interpreter to use a channel at the same time as the interpreter that created the channel (both interpreters must close the channel for the file, socket, or pipeline to actually be closed), and `interp transfer` gives access to a channel from one interpreter to another without keeping it in the source interpreter. In both cases, the channel has the same name in the target interpreter as in the source interpreter.
This can be useful in cases where you want to allow the script running in the child interpreter to open a file, but you do not want the script to know where that file is, or even that it is necessarily a file at all:

```
  set slave [interp create]
⇒ interp0
  interp alias $slave openCurrentLogfile {} openLog $slave
  proc openLog {slave} {
     global loggingHost loggingPort
     set chan [socket $loggingHost $loggingPort]
     interp transfer {} $chan $slave
     return $chan
  }
  interp eval $slave {
     set f [openCurrentLogfile]
     puts $f "script running"
     close $f
  }
```

15.11.4 Placing Limits on an Interpreter

Every interpreter has a limit on the depth of recursive evaluations it can perform.[1] This is so that procedures that accidentally call themselves in an unbounded fashion do not cause the interpreter to crash, but instead fail gracefully with a catchable error and have an informative stack trace available:

```
  proc recursive {} {
     puts "again and"
     recursive
  }
  recursive
⇒ again and
  again and
  again and
  again and
  again and
```

1. Note that there is a separate hard limit on the depth of recursion that is imposed by the underlying size of the process's stack.

```
  again and
  again and
  again and
  again and
  ...
∅ too many nested evaluations (infinite loop?)
```

The default limit is usually large enough for most programs, but some highly recursive programs require a much larger limit, and highly paranoid programmers dealing with safe interpreters may wish to set a much smaller limit. The limit can be queried and set using the interp recursionlimit command:

```
# Read the current recursion limit
set depth [interp recursionlimit $i]
# Compute the new recursion limit (half the old one)
set newDepth [expr {$depth / 2}]
# Set the new limit on the interpreter
interp recursionlimit $i $newDepth
```

A number of other limits on execution can also be enabled through the interp limit command, allowing a master interpreter to guarantee that a slave spends only a limited amount of effort carrying out some operation. The interp limit command also provides a framework for setting up traps that trigger when the limits are exceeded so that a master interpreter can decide whether to permit the slave to execute further. See the reference documentation for the interp command for more details. Still other things can be controlled relatively easily through the master interpreter, such as the number of open sockets and the total size of files being written to.

16. History

This chapter describes Tcl's history mechanism. In applications where you type commands interactively, the history mechanism keeps track of recent commands and makes it easy for you to re-execute them without having to retype them from scratch. You can also create new commands that are slight variations on old commands without having to completely retype the old commands—to fix typographical errors, for example. Tcl's history mechanism provides many of the features available in the Unix `csh`, but not with the same syntax in all cases.

Note

TkCon is a replacement for the standard console that comes with Tk. Among its many features is a sophisticated command history and editing functionality. You can use the arrow keys to cycle between commands in the history, interactively edit retrieved commands, and execute the result. TkCon also retains the command history across multiple sessions. See Appendix B for more information on TkCon and how to obtain it.

16.1 Commands Presented in This Chapter

History is implemented by the `history` command. Only a few of the most commonly used history features are described in this chapter; see the reference documentation for more complete information.

- `history`

A shortcut for `history info`.

- `history clear`

Erases the history list, resetting the event numbers. The current keep limit is retained.

- `history info ?`*`count`*`?`

Returns a formatted string giving the event number and command for each event on the history list. If *`count`* is specified, only the most recent *`count`* events are returned.

- `history keep` *`count`*

Changes the size of the history list so that the *`count`* most recent events will be retained. The initial size of the list is 20 events.

- `history nextid`

Returns the number of the next event that will be recorded in the history list.

- `history redo ?`*`event`*`?`

Re-executes the command recorded for *`event`* and returns its result.

16.2 The History List

Each command that you type interactively is entered into a *history list*. Each entry in the history list is called an *event*; it contains the text of a command plus a serial number identifying the command. The command text consists of exactly the characters you typed, before the Tcl parser performs substitutions for `$`, `[]`, and so on. The serial number starts at `1` for the first command you type and increments for each successive command.
Suppose you type the following sequence of commands to an interactive Tcl program:

```
set x 24
set y [expr {$x*2.6}]
incr x
```

At this point, the history list contains three events. You can examine the contents of the history list by invoking `history` with no arguments:

```
  history
⇒     1 set x 24
      2 set y [expr {$x*2.6}]
      3 incr x
      4 history
```

The value returned by `history` is a human-readable string describing what's on the history list, which also includes the `history` command itself. The result of `history` is intended to be printed out, not to be processed in Tcl scripts; if you want to write scripts that process the history list, you'll probably find it more convenient to use other `history` options described in the reference documentation, such as `history event`.
The history list has a fixed size, which is initially 20. If more commands than that have been typed, only the most recent commands are retained. You can change the size of the history list with the `history keep` command:

```
history keep 100
```

This command changes the size of the history list so that in the future the 100 most recent commands are retained.

16.3 Specifying Events

Several of the options of the `history` command require you to select an event from the history list. Events are specified as strings with one of the following forms:

- Positive number—selects the event with that serial number.
- Negative number—selects an event relative to the current event. `-1` refers to the last command, `-2` refers to the one before that, and so on.
- Anything else—selects the most recent event that matches the string. The string matches an event either if it is the same as the first characters of the event's command or if it matches the event's command using the matching rules for `string match`.

Suppose that you had just typed the three commands from Section 16.2. The command `incr x` can be referred to as event `-1` or `3` or `inc`, and `set y [expr {$x*2.6}]` can be referred to as event `-2` or `2` or `*2*`. If an event specifier is omitted, the default is `-1`.

16.4 Re-executing Commands from the History List

The `history redo` command retrieves a command and re-executes it just as if you had retyped the entire command. For example, after just the first three commands from Section 16.2 are typed, the command

```
history redo
```

replays the most recent command, which is `incr x`; it increments the value of variable `x` and returns its new value (`26`). If an additional argument is provided for `history redo`, it selects an event as described in Section 16.3; for example,

```
history redo 1
⇒ 24
```

replays the first command, `set x 24`.

16.5 Shortcuts Implemented by `unknown`

The basic `history` commands are quite bulky; in the previous example, it took more keystrokes to type the `history` command than to retype the command being replayed. Fortunately there are several shortcuts that allow the same functions to be implemented with fewer keystrokes:

- `!!`

Replays the last command; same as `history redo`.

- `!`*`event`*

Replays the command given by *`event`*; same as `history redo` *`event`*.

- `^`*`old`*`^`*`new`*

Retrieves the last command, replaces all occurrences of *`old`* with *`new`*, and executes the resulting command.
The last shortcut is convenient for correcting typographical errors:

```
  set x "200 illimeters"
⇒ 200 illimeters
  ^ill^mill
⇒ 200 millimeters
```

Keep in mind that Tcl's substitution shortcut replaces *all* occurrences of the old string:

```
  set x "out and about"
⇒ out and about
  ^out^in
⇒ in and abin
```

All of these shortcuts are implemented by the `unknown` procedure described in Section 15.10. `unknown` detects commands that have the forms described here and invokes the corresponding `history` commands to carry them out.

Note

If your system doesn't use the default version of `unknown` provided by Tcl, these shortcuts may not be available.

16.6 Current Event Number: `history nextid`

The command `history nextid` returns the number of the next event to be entered into the history list:

```
    history nextid
⇒ 3
```

It is most commonly used in interactive applications to generate prompts that contain the event number.

17. An Introduction to Tk

Tk is a toolkit package that allows you to create graphical user interfaces by writing Tcl scripts. Tk extends the built-in Tcl command set described in Part I with additional commands for creating user interface elements called *widgets*, arranging them into interesting layouts on the screen using *geometry managers*, and connecting them to each other with the enclosing application. Although Tk was developed to work with Tcl, other dynamic languages such as Perl, Python, and Ruby have adapted Tk to give these languages the ability to create graphical user interfaces. The commands and descriptions here apply to those languages as well, but the syntax is different. This part of the book describes Tk's Tcl commands.

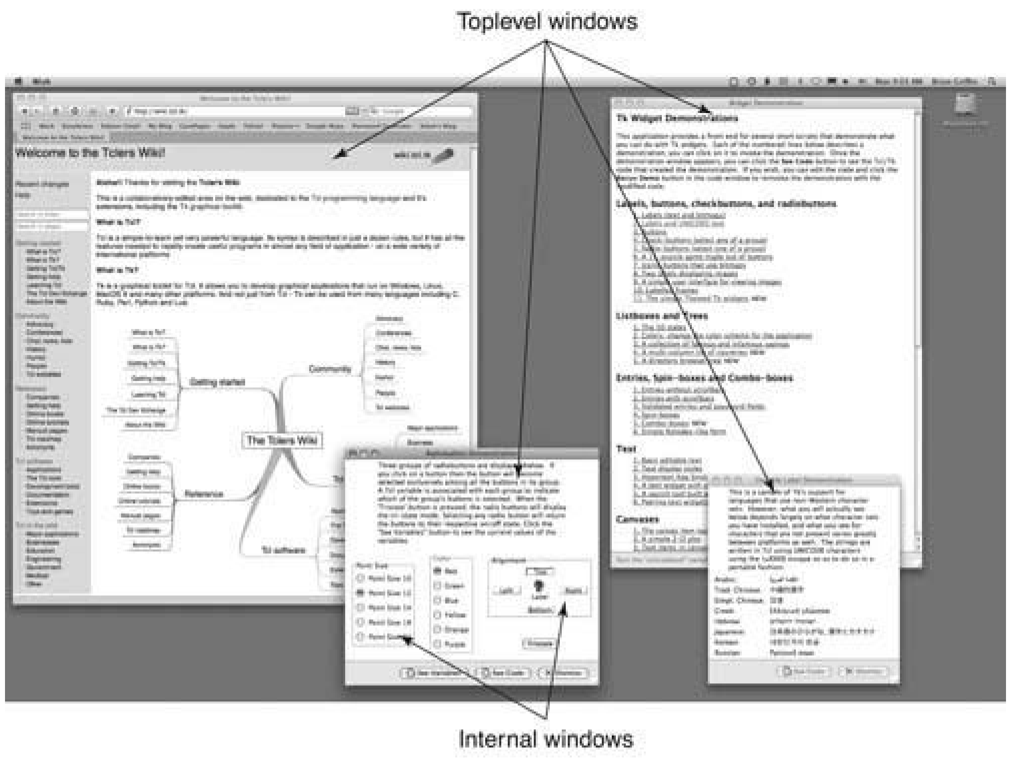
Like Tcl, Tk is implemented as a C library package that can be included in C applications, and it provides a collection of functions that can be invoked from an application to implement new widgets and geometry managers in C. The library functions are discussed in the reference documentation.

This chapter introduces the basic structures used for creating user interfaces with Tk, including the hierarchical arrangements of widgets that make up interfaces and the main groups of Tcl commands provided by Tk. Later chapters go over the individual facilities in more detail. All of the examples in this part can be run with `wish`, the windowing shell that was introduced in Chapter 1, or with any Tcl-based application by including the command `package require Tk`.

17.1 A Brief Introduction to Windowing Systems

Windowing systems provide facilities for manipulating *windows* on *displays* using a keyboard and mouse. The mouse is used to move a pointer around on the screen, and it also contains one or more buttons and possibly a scroll wheel for invoking actions. Each screen displays a hierarchical collection of rectangular windows, starting with a *root window* (usually called a desktop) that covers the entire area of the screen, as shown in Figure 17.1. The root window may have any number of child windows, each of which is called a *toplevel window*. An application typically manages several toplevel windows, one for each of the application's major panels and dialogs. Toplevel windows may have children of their own, which may also have children, and so on. The descendants of toplevel windows are called *widgets* or *internal windows*, and they may be nested to any depth. Widgets are used for individual controls such as buttons, scrollbars, or text entries and for grouping other widgets together.

Figure 17.1 Each screen contains a hierarchical collection of overlapping windows.



Events are sent to notify applications of user actions such as key presses, pointer motions, and button presses. Each event identifies what occurred (for example, the pointer just passed into a window) and provides additional information about the event such as the window where the event occurred, the time when it occurred, the position of the pointer, and the state of the mouse buttons. Events are also generated for structural changes, such as window resizes and deletions, and to notify applications that they must redraw windows, such as when one window moves so that it no longer obscures another window.

Windowing systems do not require windows to have any particular appearance or behavior. An application can draw anything it likes in any window, and it can respond to events in any way that it pleases. Thus, it is possible to implement many different appearances and modes of interaction. The lowest-level interface with a windowing system provides no support for any particular look and feel, nor does it provide built-in controls such as buttons or menus. It is up to application-level toolkits to provide these features. The Tk toolkit implements a portable library of configurable widgets that can fit into any look-and-feel environment. In this regard, Tk is unique in that it uses the native toolkit where appropriate. This allows an application to be written once in Tcl/Tk and run on multiple platforms, taking on the look and feel of the local environment.

A common element of all windowing systems is a *window manager*, also called a *desktop environment*. The window manager allows the user to manipulate toplevel windows in a uniform way for all applications. It displays a decorative frame around each toplevel window and provides controls within the frame so that users can interactively move, resize, iconify, and deiconify windows. The decorative frame also displays a title for the toplevel window. Many different window managers exist. On some systems, like Mac OS X and Microsoft Windows, there is only one by default, but on Unix and Linux systems, there are many to choose from. Each window manager may have different decorations and controls, but only one window manager exists for a given display at a given time. The examples in this book use a variety of window managers or desktop environments to demonstrate Tk's versatility and portability in all these environments.

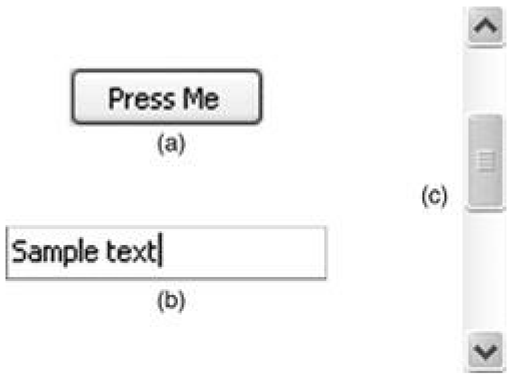
Applications must communicate with both the windowing system and the window manager. The application communicates with the windowing system to create windows, draw on them, and receive events. It communicates with the window manager to specify titles, preferred dimensions, and other information for its toplevel windows.

The three primary windowing systems that Tk supports are X, for Unix and Linux operating systems; Microsoft Windows; and Apple's Mac OS X, also known as Aqua. For additional information on these windowing systems, search the web for *X.org*, *Microsoft User Interface Design and Development*, or *Apple Human Interface Guidelines*, respectively.

17.2 Widgets

Tk uses low-level drawing APIs to implement a ready-made set of controls, which are commonly called *widgets*. Figure 17.2 shows examples of button, entry, and scrollbar widgets. Each widget is a member of a *class* that determines its appearance and behavior. For example, widgets in the `button` class display a text string or image. Different buttons may display their strings or images in different ways (for example, with different fonts and colors), but each one displays a single string, image, or combination. Each button also has a Tcl script associated with it, which is invoked if mouse button 1 (the left button on a three-button mouse) is pressed and released when the pointer is over the widget. Different button widgets may have different scripts associated with them to implement different operations. When you create a widget, you select its class and provide additional class-specific *options*, such as a string or image to display or a script to invoke.

Figure 17.2 Examples of widgets: (a) a button widget; (b) an entry widget; (c) a scrollbar widget



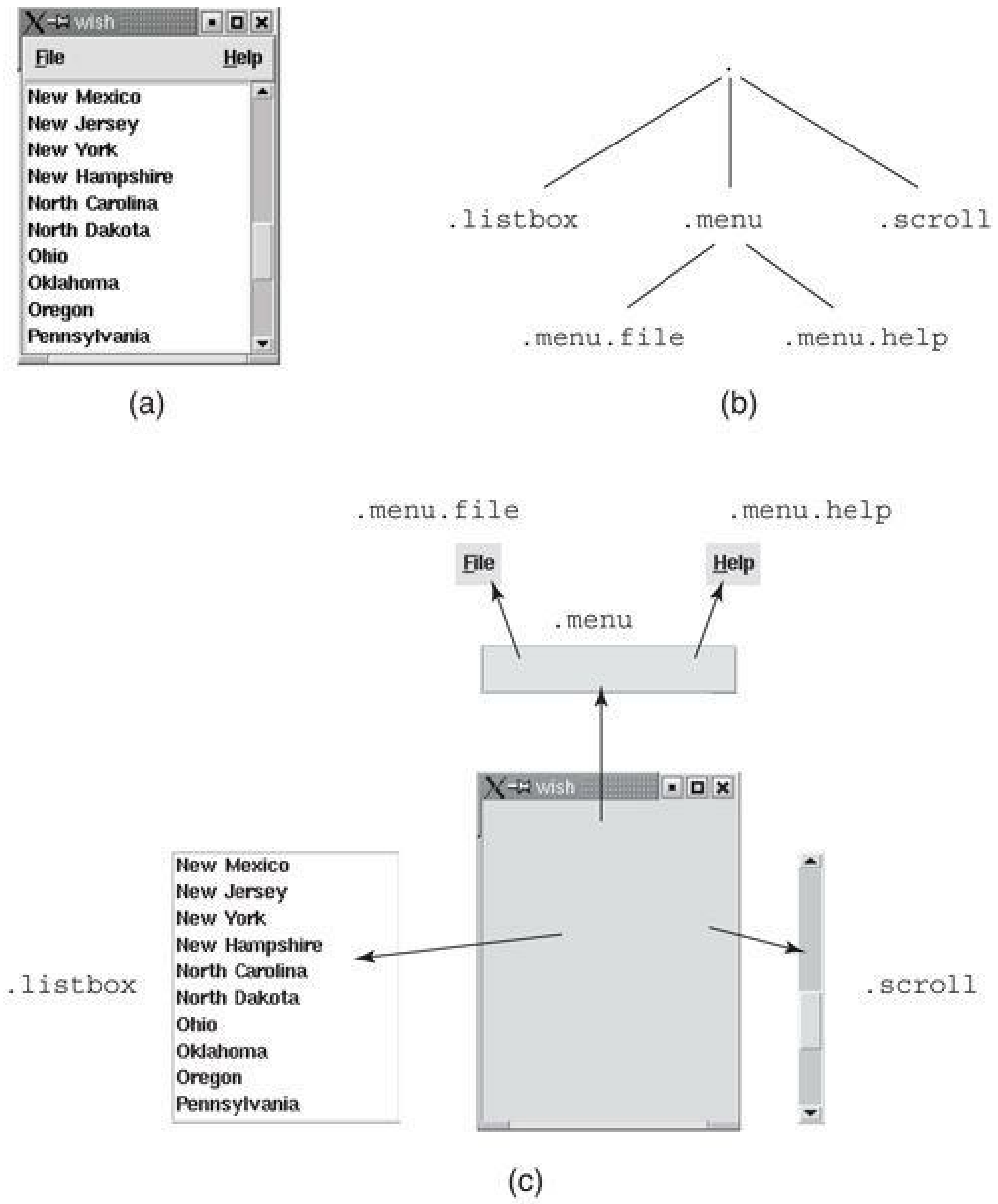
Each Tk widget is implemented using one window.[1] As with windows, widgets are nested in hierarchical structures. A widget can contain any number of children, and the widget tree can have any depth. Widgets such as buttons that have meaningful behavior for the user are usually at the leaves of the widget tree; the non-leaf widgets are usually just containers for organizing and arranging the leaf widgets.

1. The terms *widget* and *window* are used interchangeably in this book and in the reference documentation.

Each widget or window has a textual name such as `.a.b.c`. Widget names are similar to the hierarchical path names for files, except that `.` is used as the separator character instead of `/` or `\`. The name `.` refers to the topmost widget in the hierarchy, which is called the *main window*. The name `.a.b.c` refers to a widget `c` that is a child of widget `.a.b`, which in turn is a child of `.a`, which is a child of the main window.
Figure 17.3 shows how a window is composed. The window is shown in (a) as it appears on the screen, with decorations provided by the windowing system. The hierarchical structure of the widgets is shown in (b). An exploded view of the screen is shown in (c) to clarify the widget structure.

The topmost widget in the hierarchy (.) contains three children: a menu across the top, a scrollbar along the right side, and a listbox filling the remainder. The menu bar contains two children of its own, a File menu on the left and a Help menu on the right. Each widget has a name that reflects its position in the hierarchy, such as `.menu.help` for the Help menu.

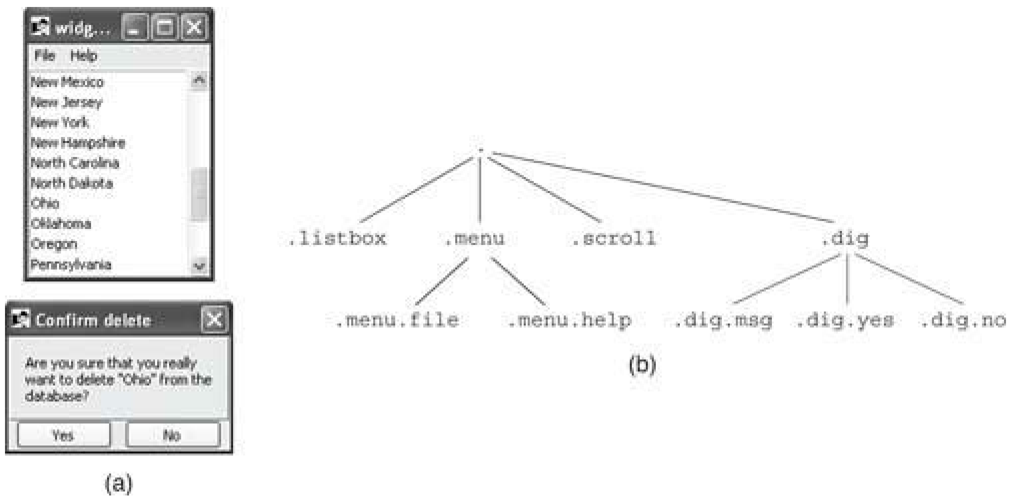
Figure 17.3 Window composition



17.3 Applications, Toplevel Widgets, and Screens

In Tk the term *application* refers to a single widget hierarchy (a main widget and its descendants), a single Tcl interpreter associated with the widget hierarchy, plus all the commands provided by that interpreter. Each application has its own widget hierarchy, so the name `.` refers to the main window or *root window* within the application. Normally an application runs in a single process, but Tk also allows a single process to manage several applications with separate widget hierarchies, one per Tcl interpreter. Tk does not require or provide any particular support for multithreading but may be used in a multithreaded environment, provided that all Tk commands and operations for a given window hierarchy are handled in a single thread and interpreter.
The main widget for each application occupies a toplevel window, so it is decorated and managed by the window manager. Most of the other widgets of an application use internal windows. The `toplevel` widget class is used to create additional toplevel windows so that the window manager can manipulate them independently. A toplevel widget differs from an internal widget in that its window is a child of the root window for its screen, whereas the window for an internal widget is a child of the window for the widget's parent in the Tk hierarchy. Toplevel widgets are typically used as containers for panels and dialogs, as shown in Figure 17.4. In this example the dialog box `.dlg` is a toplevel widget, as is the main widget. Figure 17.4(a) shows how the widgets appear on the screen, and (b) shows Tk's widget hierarchy for the application.

Figure 17.4 Toplevel widgets



Are you sure that you really want to delete "Ohio" from the database?

17.4 Scripts and Events

A Tk application is controlled by two kinds of Tcl scripts: an *initialization script* and *event handlers*. The initialization script is executed when the application starts. It creates the application's user interface, loads the application's data structures, and performs any other initialization needed by the application. Once initialization is complete, the application enters an *event loop* to wait for user interactions. Whenever an event occurs, such as the user invoking a menu entry or moving the mouse, a Tcl script is invoked to process that event. These scripts are called event handlers; they can invoke application-specific Tcl commands to enter an item into a database, modify the user interface (for example, by posting a dialog box), and do many other things. Some event handlers are created by the initialization script, but event handlers can also be created and modified by other event handlers; for example, an event handler for a menu might create a new dialog box along with new event handlers for the dialog.
Most of the Tcl code for a Tk application is in the event handlers. Complex applications may contain hundreds of event handlers, and the handlers may create other panels and dialogs that have additional event handlers. Tk applications are therefore *event-driven*. There is no well-defined flow of control within the application's scripts, since there is no single task for the application to perform. The application presents a user interface with many features, and the user decides what to do next. All the application does is respond to the user's actions. The event handlers implement the responses; event handlers tend to be short, and they are mostly independent of each other.

17.5 Creating and Destroying Widgets

Tk provides four main groups of Tcl commands: for (1) creating and destroying widgets, (2) arranging widgets on the screen, (3) communicating with existing widgets, and (4) interconnecting widgets within and between applications. This section and the three following sections introduce the groups of commands to give you a general feel for Tk's features. All of the commands are discussed in more detail in later chapters.
To create a widget, you invoke a command named after the widget's class: `button` for button widgets, `scrollbar` for scrollbar widgets, and so on. These

commands are called *class commands*. For example, the following command creates a button that displays the text `Press me`:

```
button .b -text "Press me" -command {puts Ouch!}
```

All class commands have a form similar to this. The command’s name is the name of a widget class. The first argument is the path name for the new widget, `.b` in this case. The command will create the widget and its corresponding window. The widget name is followed by any number of pairs of arguments, where the first argument of each pair specifies the name of a *configuration option* for the widget, such as `-text` or `-command`, and the second argument specifies a value for that option, such as `Press me` or `{puts Ouch!}`. Each widget class supports a different set of configuration options, but many options, such as `-foreground`, are used in the same way by different classes. Values need not be specified for every option supported by a widget; Tk provides defaults for the unspecified options. For example, buttons support about 35 different options but only 2 were specified in the preceding example. Chapter 18 describes many of the most common configuration options.
To delete a widget, you invoke the `destroy` command:

```
destroy .menubar
```

This command destroys the widget named `.menubar` and all of its descendants, if there are any.

17.6 Geometry Managers

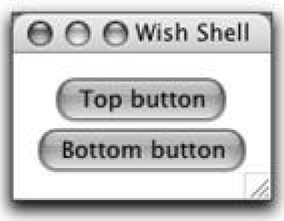
Widgets don’t determine their own sizes and locations on the screen; *geometry managers* carry out this function. Each geometry manager implements a particular style of layout. Given a collection of widgets to manage and some controlling information about how to arrange them, a geometry manager assigns a size and location to each widget. For example, you might tell a geometry manager to arrange a set of widgets in a vertical column. The geometry manager then positions the widgets so that they abut and do not overlap. If a widget should suddenly need more space (for example, because its font was changed to a larger one), the widget notifies the geometry manager and the geometry manager moves other widgets to preserve the column structure.

The second of the four main groups of Tk commands consists of those for communicating with geometry managers. Tk currently contains three geometry managers plus three widgets that can manage embedded widgets. The main geometry manager for Tk is the *grid manager*, which works by arranging widgets into rows and columns. The other primary geometry managers in Tk are the *packer*, which works by packing a series of widgets sequentially around the edges of a cavity, and the *placer*, which provides simple fixed or relative placements. Additionally, the *window manager* is used to manage toplevel windows in conjunction with the desktop environment. The canvas, text, and panedwindow widgets have an internal geometry manager for embedded widgets. Chapter 21 describes the geometry managers in more detail.

When you invoke a class command like `button`, the new widget does not appear immediately on the screen. It appears only after you ask a geometry manager to manage it. If you want to experiment with widgets before reading the full discussion of geometry managers, you can make a widget appear by invoking the `grid` command with the widget's name as its argument. This sizes the widget's parent so that it is just large enough to hold the widget and arranges the widget so that it fills the space of its parent. If you create other widgets and grid them in a similar fashion, the grid manager arranges them in a column inside the parent, making the parent just large enough to accommodate them all, as shown in Figure 17.5. The following script creates two button widgets and arranges them in a vertical column with the first widget above the second:

```
button .top -text "Top button"
button .bottom -text "Bottom button"
grid .top
grid .bottom
```

Figure 17.5 Arranging widgets in a window



17.7 Widget Commands

Whenever a new widget is created, Tk also creates a new Tcl command with the same name as the widget. This command is called a *widget command*, and the set of all widget commands (one for each widget in the application) constitutes the third major group of Tk commands. After the `button` command was executed in Section 17.5, a widget command whose name is `.b` appeared in the application's interpreter. This command exists as long as the widget exists; when the widget is deleted, the command is deleted, too.
Widget commands are used to communicate with existing widgets. Here are some commands that could be invoked after the `button` command from Section 17.5:

```
.b configure -command runCommand
.b invoke
```

The first command changes the callback made when the button is pressed, and the second command invokes the button's command as if the user had clicked mouse button 1 over the widget. In widget commands, the command name is the name of the widget, and the first argument specifies an *action* to invoke on the widget. Some actions, like `configure`, take additional arguments; the nature of these arguments depends on the specific action.

Note

The terms *action*, *widget command*, and *widget subcommand* are used interchangeably throughout this book. For example, the phrase "the `configure` widget command" really means "the `configure` action for the widget command."

The set of widget commands supported by a given widget is determined by its class. All widgets in the same class support the same set of widget commands, but different classes have different command sets. Some common actions are supported by multiple classes. For example, every widget class supports a `configure` widget command, which can be used to query and change the widget's configuration options.

17.8 Commands for Interconnection

The fourth group of Tk commands is used for interconnection. These commands are used to make widgets work together, to make them work cooperatively with the objects defined in the application, and to allow different applications sharing the same display to work together in interesting ways.

Some of the interconnection commands are implemented as event handlers. For example, each button has a `-command` option that specifies a Tcl script to invoke whenever mouse button 1 is clicked over the widget. Scrollbars provide another example of interconnection via event handlers. Each scrollbar is used to control the view in some other widget. When you click in the scrollbar or drag its slider, the view in the associated widget should change. This connection between widgets is implemented by specifying a Tcl command for the scrollbar to invoke whenever the slider is dragged. The command invokes a widget command for the associated widget to change its view. Other kinds of event handlers and event generation are described in Chapter 22.

Some of the Tk widgets support a model-view-controller paradigm via a link to a Tcl variable. The value of the variable is used to define the widget's text content or state. The checkbutton and radiobutton widgets actually require it for the widgets to work correctly. This feature is discussed in more detail in Chapter 18; also see the reference documentation for information on the `-textvariable` and `-variable` options provided on several widgets.

Another form of interconnection is selection management. The selection is a highlighted piece of information on the screen, such as a range of text or a graphic. Tk provides a way to manage the selection in conjunction with the window manager so that applications can claim ownership of the selection and retrieve its contents from whichever application owns it. Chapter 25 discusses the selection in more detail and describes Tk's `selection` command. Chapter 26 describes Tk's window manager command, `wm`, which is used for communicating with the window manager. The window manager acts as a geometry manager for toplevel windows, and the `wm` command can be used to make specific geometry requests of the window manager, such as "Don't let the user make this window smaller than 80 pixels tall." In addition, `wm` can be used to specify a title to appear in the window's decorative border, a title and/or icon to display when the window is iconified, and many other attributes.

At any given time the keystrokes typed for an application are directed to a particular widget, regardless of the mouse cursor's location. This widget is referred to as the *focus widget* or the widget having *focus*. Chapter 27 describes the `focus` command, which is used to query about the current focus widget or to change the focus. Chapter 27 also describes *grabs*, which restrict keyboard and mouse events so that they are processed only in a subtree of the widget hierarchy. Windows outside the grab subtree are blocked from receiving any events until the grab is released. Grabs are used to disable parts of an application and force the user to deal immediately with a high-priority window such as a dialog box.

Finally, Chapter 12 describes several ways for programs to communicate. The `send` command is one way, specific to Tk, providing a general-purpose means of communication between applications. You can use `send` to issue an arbitrary Tcl command to any Tk application on the display.

18. A Tour of the Tk Widgets

This chapter introduces the classic widget classes implemented by Tk, which have been around since nearly the beginning of Tcl. They provide a cross-platform means to create user interfaces for your scripts. In addition, these classic widgets provide an easy way to add a user interface to an existing application, whether written in Tcl or not. With the ability to embed a Tcl interpreter within applications, you can extend your applications with a graphical user interface. Or you can use Tcl's commands to drive a command-line application and use Tk widgets to introduce a user interface.

In addition to the classic Tk widgets, recent years have seen the development of a set of *themed widgets*. Themes aim to separate widget functionality from the visual look of the widgets. In addition, themes allow for much better-looking widgets. The classic Tk look comes from the early era of Windows 95 and Motif on Unix; the themed widgets provide a more modern appearance, as well as a unified and modifiable look and feel. This chapter introduces the older classic Tk widgets. The next chapter covers the themed widgets.

The main advantage to using themed widgets is to give a native look to a Tk application by selecting the style that fits the user's current desktop environment. While Tk uses native widgets on Microsoft Windows and Mac OS X for scrollbars, checkbuttons, and radiobuttons, X-based classic Tk widgets have the basic Motif look. The themed widgets provide a variety of styles that match a variety of desktop environments available on Unix operating systems. New styles can also be added to make a themed application instantly fit into new desktop environments. The newer themed widgets are still evolving and don't yet provide the full functionality of the older widgets. There are also some classic Tk widgets that don't have a themed counterpart. However, you can combine classic and themed widgets in the implementation of your application's interface.

Note

As of Tk 8.5, the widget class commands for creating classic Tk widgets are defined in the `::tk` namespace; for example, `::tk::button` creates a classic Tk button widget. In contrast, the widget class commands for creating themed widgets are defined in the `::ttk` namespace. For backward compatibility with existing scripts, the classic Tk widget class commands are imported automatically into the global namespace. This may be discontinued in a future release of Tk, in which case you could add a `namespace import` command to your existing scripts to explicitly import the needed widget class commands. See Chapter 10 for more information on namespaces.

Note

The examples in this chapter use the `grid` command to arrange widgets, though you don't need to understand the operation of the `grid` commands yet in order to understand the examples. Chapter 21 discusses the `grid` command in detail.

18.1 Widget Basics

Each classic Tk widget class is defined by three things: its *configuration options*, its *widget commands,* and its *default bindings*. Configuration options represent most of the state information for most widgets; they include things such as the colors, fonts, and text to display in a button widget and the Tcl script to invoke when the user clicks on the widget. All of the widgets in a class support the same configuration options. Different classes may have different options, but many options, such as `-foreground` and `-font`, are supported by many different classes. The configuration options for a widget may be specified when the widget is created and modified later with the `configure` widget command. Default values can be specified using the option database (see Chapter 28).
The second thing that defines a widget class is its widget command. As described in Section 17.7, one widget command exists for each widget in an application, and it is used to invoke actions in the widget. The actions provided by a widget command vary from class to class, but some actions are supported by several classes. For example, every class supports a `configure` action for changing the widget's configuration options. For more complex widgets such as menus and listboxes, the widget's internal state is

too complex to represent entirely with configuration options, so additional actions are provided to manipulate the state. For example, the widget command for menus provides actions to create and delete entries and to modify existing entries.

The third thing that defines a widget class is its default bindings. The default bindings give widgets their behavior by causing Tcl scripts to be evaluated in response to user actions. Tk creates the default bindings by evaluating an initialization script stored in the Tk library directory; the script invokes the `bind` command described in Chapter 22. This means that the behaviors are not hard-coded into the widgets; you can modify the behavior of individual widgets or entire classes using the `bind` command. The descriptions in this chapter correspond to the default behaviors.

Tk currently defines the widget classes listed in Table 18.1. In addition to these classic widgets, the themed widget extension provides 18 themed widgets (see Chapter 19). The themed widgets provide additional functions and an updated look and feel, so when there is a choice, these widgets should be preferred over the classic Tk widgets for new application development.

Table 18.1 Classic Tk Widget Classes

`button`	`canvas`	`checkbutton`	`entry`	`frame`
`label`	`labelframe`	`listbox`	`menu`	`menubutton`
`panedwindow`	`radiobutton`	`scale`	`scrollbar`	`spinbox`
`text`	`toplevel`			

This chapter gives an overview of most of the classic widget classes along with the most commonly used configuration options; canvas and text widgets are described separately in Chapter 23 and Chapter 24, respectively. This chapter does not explain every feature of every widget; for that you should refer to the reference documentation for the individual widget classes. If you wish to see additional examples of widget usage besides those in this chapter, you should run the demonstration scripts in the Tk distribution. In particular, the script `widget` contains examples of various widgets. You can run the `widget` demo using the following commands from a `wish` interpreter:

```
cd $tk_library
cd demos
source widget
```

The `widget` script creates a window with many links to examples of various

Tk widgets. For each example, you can see a live demonstration of the widget as well as view the example source code.

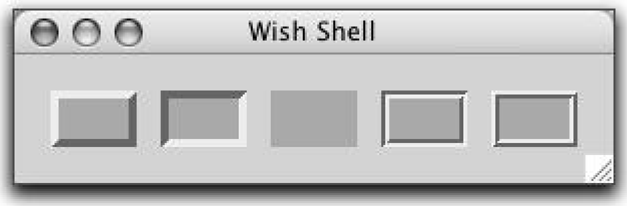
18.2 Frames

Frames are the simplest widgets; they appear as colored rectangular regions and can have three-dimensional borders. As you will see later, frames typically serve as containers for grouping other widgets. Most of the non-leaf widgets in an application are frames. Frames are particularly important for building nested layouts with geometry managers; when used in this way, frames are often invisible to the user. This is discussed in Chapter 21. Frames are also used to generate decorations such as a block of color or a raised or sunken border around a group of widgets. Frames have no default behavior; they do not normally respond to the mouse or keyboard.

18.2.1 Relief Options

Frames support only a few configuration options, and most of these are supported by all basic widget classes. For example, the `-relief` and `-borderwidth` options can be used to specify a three-dimensional border. The `-relief` option determines the appearance of the border and must have one of the values `raised`, `sunken`, `flat`, `groove`, or `ridge`. Tk draws widget borders with light and dark shadows to produce the different effects. For example, if a widget's relief is `raised`, Tk draws the top and left borders in a lighter color than the widget's background and it draws the lower and right borders in a darker color. This makes the widget appear to protrude from the screen. An example of each `-relief` setting is shown in Figure 18.1.

Figure 18.1 Five frames with different relief configurations



The source for this example follows. Each frame sports a darker color to help distinguish the frame boundaries.

```
set c -1
foreach relief {raised sunken flat groove ridge} {
    frame .$relief -width 15m -height 10m -relief $relief \
    -borderwidth 4 -background {dark grey}
    grid .$relief -column [incr c] -row 0 -padx 2m -pady 2m
}
. configure -background {light grey} -padx 12 -pady 12
```

The three-dimensional appearance of a theme widget comes from the theme, so you normally don't need to set the relief setting on `ttk::frame` or other themed widgets.

18.2.2 Screen Distance Options

Several commonly used configuration options take *screen distances* as values. For example, the `-borderwidth` option controls the border width, the `-width` and `-height` options can be used to specify the dimensions of a frame, and widgets such as text provide an `-insertwidth` option for specifying the size of the insertion cursor. Screen distances can be specified either in pixels or in absolute units independent of the screen resolution. A distance consists of an integer or floating-point value followed optionally by a single character giving the units. If no unit is specified, the units are pixels. Otherwise the unit is designated by one of the following characters:

- `c`—centimeters
- `i`—inches
- `m`—millimeters
- `p`—printer's points (1/72 inch)

For example, a distance specified, as `2.2c` is rounded to the number of pixels that most closely approximates 2.2 centimeters; the number of pixels may vary from screen to screen. The code that produced Figure 18.1 contains examples of screen distance options specified in millimeters and pixels.
Screen distances are used for other purposes in Tk besides widget configuration options. One example is the `-padx` and `-pady` options to the `grid` command, which are used in the code that produced Figure 18.1 to leave extra space around each of the frame widgets.

18.3 Color Options

The `-background` option determines the background color of a widget and also the shadow colors used in its border. The value of the `-background` option may be specified either symbolically or numerically. A symbolic color value is a name such as `white` or `red` or `SeaGreen2`. Tk defines a large number of symbolic color values available on all platforms based on standard Unix color values. Color names are not case-sensitive: `black` is the same as `Black` and `bLaCk`. Refer to the `colors` page of the reference documentation for a list of valid color names. Windows and Mac OS X platforms also support a set of aliased colors, such as `ActiveBorder` and `systemWindowBody`, which refer to user preference settings in these environments. These symbolic color values are also documented in the `colors` reference documentation.
Colors can also be specified numerically in terms of their red, green, and blue components. Four forms are available, in which the components are specified with 4-bit, 8-bit, 12-bit, or 16-bit values:

```
#RGB
#RRGGBB
#RRRGGGBBB
#RRRRGGGGBBBB
```

Each `R`, `G`, or `B` in these examples represents one hexadecimal digit of red, green, or blue intensity, respectively. The first character of the specification must be `#`, and the same number of digits must be provided for each component. If fewer than four digits are given for each component, they represent the most significant bits of the values. For example, `#3a7` is equivalent to `#3000a0007000`. A value of all `1`s represents "full on" for that color, and a value of `0` represents "off." Thus, `#000` is black, `#f00` is red, `#ff0` is yellow, and `#fff` is white.

Note

If you specify a color other than black or white for a monochrome display, Tk uses black or white instead, depending on the overall intensity of the color you requested. Furthermore, if you are using a color display and all of the entries in its color map are in use (e.g., because you're displaying a complex image on the screen), Tk treats the display as if it were monochrome.

Virtually every basic widget class supports a `-background` option, and most widget classes support additional color options. For example, most widget classes provide a `-foreground` option that determines the color of the text or graphics displayed in the widget. For theme widgets, the color options are defined by the theme.

18.3.1 Synonyms

Tk provides special short forms for a few of the most commonly used options. For example, `-bd` is a synonym for `-borderwidth`, `-bg` is a synonym for background, and `-fg` is a synonym for foreground.

18.4 Toplevels

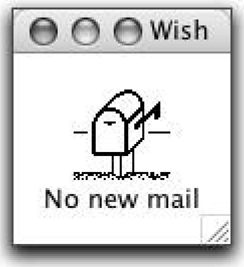
Toplevel widgets are identical to frames except that they occupy toplevel windows, whereas frames occupy internal windows. Toplevels are typically used as the outermost containers for an application's panels and dialog boxes. The main widget for an application is also a toplevel. When creating a new toplevel widget, you can use the `-menu` option to specify a menu widget to use as that window's menu bar. The menu bar is displayed along the top of the window as part of the window manager's decoration. On Mac OS X, the menu bar is displayed across the top of the screen. Chapter 26 provides more information on managing toplevels.

18.5 Labels

A label widget displays information to the user. A label widget may display a text string, a bitmap, an image, or a combination of text and a bitmap or image. Labels provide several additional configuration options for specifying what to display in the widget; Figure 18.2 is an example of a compound label generated by the following script:

```
label .label -text "No new mail" -bitmap \
    @$tk_library/demos/images/flagdown.xbm \
    -compound top
grid .label
```

Figure 18.2 The label widget displaying a bitmap and a text string



18.5.1 Text Options

Most widgets that display simple text strings, like labels, provide two options for specifying the string. If a `-text` option is specified, as for `.label` in the script for Figure 18.2, its value is the string to display in the widget. If instead you specify the `-textvariable` option, you can set its value to the name of a variable referenced from the global namespace. The `-textvariable` option causes the widget to display the contents of the variable; whenever the variable changes, the widget resizes and/or redisplays itself to reflect the new value.
The text string displayed by the label can contain embedded newline characters, in which case the label displays multiple lines of text. The `-justify` option determines how the lines align with each other, and you can set it to `left`, `center`, or `right`.

Note

A common practice is to use namespace variables in `-textvariable` references to prevent polluting the global namespace with too many variables. This requires using fully qualified namespace names (e.g., `::Application::country`).

Here is a simple example that uses `-textvariable`:

```
proc watch {name} {
    toplevel .watch
    label .watch.label -text "Value of \"$name\": "
    label .watch.value -textvariable $name
    grid .watch.label .watch.value -pady 12
}
```

This procedure will create a new toplevel widget and grid two labels inside it to display the name of a global variable and its value. For example, the following script can be invoked to create the panel shown in Figure 18.3(a):

```
set country Japan
watch country
```

Figure 18.3 The `-textvariable` option ties a widget's value to a variable.

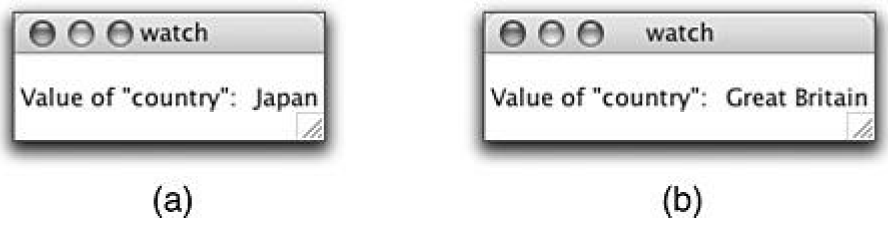


Figure 18.3(b) shows how the display changes if the following command is invoked to change the variable's value:

```
set country "Great Britain"
```

18.5.2 Font Options

The `-font` option specifies a font to use when displaying text in a widget. Tk uses a standard font description of *`family size style`*. . ., where *`size`* and *`style`* are optional. Some example font descriptions are

```
{Times 12 bold italic}
{Helvetica 14 bold}
{Courier 10 normal}
```

Other description formats are supported as well, including standard X11 font descriptions. The Tk `font` command can be used to create and manage application fonts; it is discussed in more detail in Chapter 20.

Note

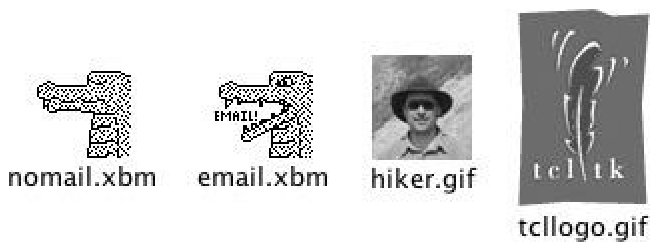
For almost all widget classes, all characters within a widget use the same font and style. The text widget, on the other hand, can display multiple lines of text and apply different fonts and other display characteristics to substrings. See Chapter 24 for more information on the text widget.

18.5.3 Image Options

Labels and many other widgets can display images in the form of *bitmaps* or *images* instead of text. A bitmap is a picture with two colors, foreground and background. Bitmaps are specified using the `-bitmap` option, whose values may have two forms. If the first character of the value is `@`, the remainder of the value is the name of a file containing a bitmap in X bitmap file format. Thus `-bitmap @face.xbm` specifies a bitmap contained in the file `face.xbm`. The script for Figure 18.2 specifies a bitmap as `@$tk_library/demos/images/flagdown.xbm`. This refers to one of the bitmaps in the library of demonstrations that is included with the Tk distribution (the variable `tk_library`'s library files).
If the first character of the value isn't `@`, the value must be the name of a bitmap defined internally. Tk defines a few internal bitmaps itself that are used in message dialogs, and individual applications may define additional ones.
The `-bitmap` option determines only the pattern of `1`s and `0`s that make up the bitmap. The foreground and background colors used to display the bitmap are determined by the `-foreground` and `-background` options for the widget. This means that the same bitmap can appear in different colors at different places in an application, and the colors of a given bitmap may be changed by modifying the `-foreground` and `-background` options.
Images are created using the Tk `image create` command. Some example images are shown in Figure 18.4. Images are objects created from image data directly, or from various file formats. Tk directly supports the XBM, XPM, GIF, and PPM/PGM formats. Tk extensions such as Img and TkMagick provide support for just about all other image file formats commonly in use.
Images are specified using the `-image` option as shown here:

```
image create photo hiker -file hiker.gif
toplevel .images
label .images.l -text hiker -image hiker -compound top
grid .images.l
```

Figure 18.4 Sample images using the Tk `image` command



Images are shared among all widgets that reference them. Any change to an image is reflected everywhere it appears on the screen. Images are discussed in more detail in Chapter 20 on fonts and images.

18.5.4 Compound Options

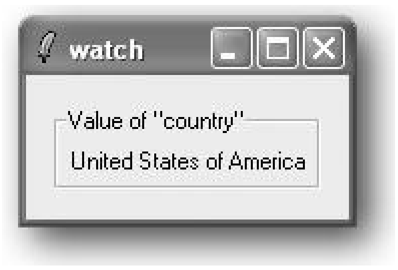
Labels may display both an image and text, as shown in Figure 18.2. The position of the image relative to the text is specified with the `-compound` option. The values `top`, `bottom`, `left`, `right`, and `center` will place the image above, below, left of, right of, and centered on the text respectively. A value of `none` displays only the image if an `-image` or `-bitmap` option is used; otherwise only the text is displayed.

18.6 Labelframes

While frames are used to organize widgets into groups, the labelframe widget provides all the features of the frame widget plus, as the name suggests, a label and a border ring, as shown in Figure 18.5. The label may be placed in any of the eight compass directions around the frame using the `-labelanchor` option. The labelframe is a common element found in dialogs and panels. The following script generated the labelframe figure shown:

```
proc watch {name} {
    toplevel .watch -padx 14 -pady 15
    labelframe .watch.lf -text "Value of \"$name\""
    label .watch.lf.value -textvariable $name
    grid .watch.lf.value -padx 3 -pady 3
    grid .watch.lf -sticky nsew
}
set country "United States of America"
watch country
```

Figure 18.5 An example of a labelframe widget



Value of "country"

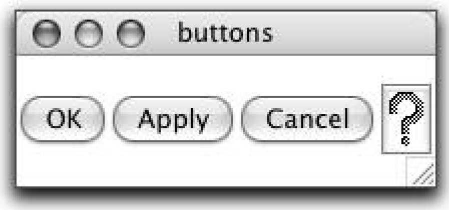
United States of America

18.7 Buttons

Buttons, checkbuttons, radiobuttons, and menubuttons make up a family of widget classes with similar characteristics. These classes have all of the features of labels, and they also respond to the mouse. When the pointer moves over a button, the button lights up to indicate that something will happen if a mouse button is pressed; a button in this state is said to be *active*. It is a general property of Tk widgets that they light up if the pointer passes over them when they are prepared to respond to button presses. Buttons become inactive when the pointer leaves them.
If mouse button 1 is pressed when a button widget is active, the widget's appearance changes to make it look sunken, as if a real button had been pressed. When the mouse button is released, the widget's original appearance is restored. Furthermore, when the mouse button is released, the widget evaluates its `-command` option as a Tcl script. Figure 18.6 is an example that can be created with a script like the following:

```
button .ok -text OK -command
button .apply -text Apply -command
button .cancel -text Cancel -command cancel
button .help -text Help -bitmap question -command help
grid .ok .apply .cancel .help
```

Figure 18.6 Four button widgets

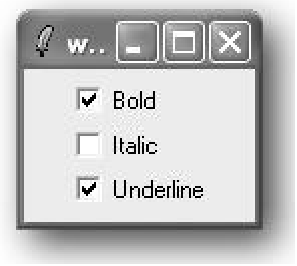


The procedures `ok`, `apply`, `cancel`, and `help` are invoked when the user clicks on one of the buttons with the mouse; the code for these procedures is not provided here. In this example the `-command` options are all single words, but in general they can be arbitrary scripts.

18.7.1 Checkbuttons

Checkbuttons are used for making binary choices such as enabling or disabling underlining or grid alignment; see Figure 18.7 for an example. Checkbuttons are similar to buttons except for two things. First, when you click mouse button 1 over a checkbutton, a global variable toggles in value between `0` and `1`. The variable name is specified to the widget with the `-variable` option. Second, the checkbutton displays a rectangular *selector* to the left of its text or bitmap. When the variable's value is `1`, the selector is displayed as a check mark and the checkbutton is said to be *selected*; otherwise the selector box appears empty. Each checkbutton monitors the value of its associated variable, and if the variable's value changes (e.g., because of a `set` command), the checkbutton updates the selector display. Checkbuttons provide additional configuration options for specifying the color of the selector and for specifying "off" and "on" values other than `0` and `1`.

Figure 18.7 Three checkbuttons



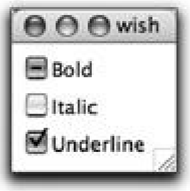
The buttons in Figure 18.7 are created by the following script. Each checkbutton toggles a particular variable between `0` and `1`, and its selector square indicates the current value of the variable (the variables `bold` and `underline` are currently `1`, `italic` is `0`). The `-anchor` options cause the buttons to left-justify their text and selectors as described in Section 18.15.3.

```
checkbutton .bold -text Bold -variable bold -anchor w
checkbutton .italic -text Italic -variable italic -anchor w
checkbutton .underline -text Underline \
        -variable underline -anchor w
grid .bold -sticky ew
grid .italic -sticky ew
grid .underline -sticky ew
```

In some situations a checkbutton may be used to represent the state of multiple items. In these cases it is useful for the widget to display a state that represents both "on" and "off." Take the preceding example where the `bold`, `italic`, and `underline` state of some text is represented. If the text had a mix of bold and normal characters, the button would need to represent both states. This is done using the *tristate* value for the widget, as shown in Figure 18.8. The tristate value causes the widget to display an alternate mark that represents this "mixed" condition. You specify the tristate value with the widget's `-tristatevalue` option:

```
.bold configure –tristatevalue 3
set bold 3
```

Figure 18.8 Example checkbutton in "tristate" or "mixed" state

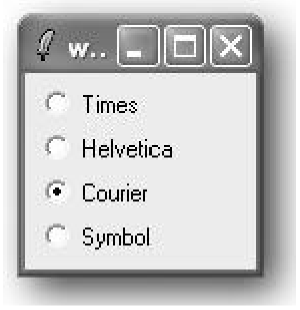


18.7.2 Radiobuttons

Radiobutton widgets provide a way to select one of several mutually exclusive options, much as the radio buttons in cars are used to select one of several stations. Several radiobuttons work together to control a single global variable. Each radiobutton provides a `-variable` option to name the variable and a `-value` option to specify a value for the variable; when you click on the widget, it sets the variable to the given value. For example, Figure 18.9 shows a collection of radiobuttons that are used to select a font family from among several alternatives. Each radiobutton displays a round selector to the left of its text or bitmap and lights up the selector when it is selected (i.e., whenever the variable's value matches its `-value` option). Each radiobutton monitors the variable and turns its selector on or off when the variable value changes. The user can select one of four values for the global variable `font` by clicking on a button. In the figure the selector circle for the `.courier` widget is marked to indicate that it is selected (the variable currently has the value `courier`).

```
radiobutton .times -text Times -variable font \
    -value times -anchor w
radiobutton .helvetica -text Helvetica -variable font \
    -value helvetica -anchor w
radiobutton .courier -text Courier -variable font \
    -value courier -anchor w
radiobutton .symbol -text Symbol -variable font \
    -value symbol -anchor w
grid .times -sticky ew
grid .helvetica -sticky ew
grid .courier -sticky ew
grid .symbol -sticky ew
```

Figure 18.9 Four radiobuttons



The radiobutton also supports a tristate value and display, similarly to the checkbutton, as shown in Figure 18.10.

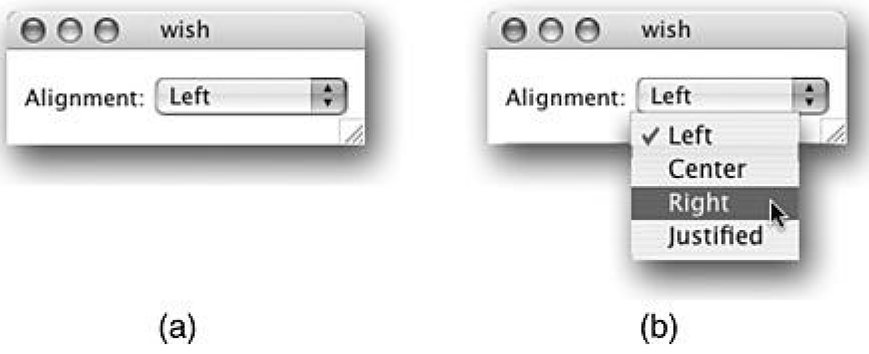
Figure 18.10 The tristate appearance of the radiobutton widget

18.7.3 Menubuttons

The menubutton widget creates a button that, when pressed, posts a menu. Menus are discussed in more detail in Section 18.12. Like the other button widgets, the menubutton can be configured with any combination of text and image and also has an indicator. Figure 18.11 demonstrates a menubutton used to select an alignment option by using radiobutton menu entries. The following script creates this menubutton:

```
menubutton .mb -width 9 -textvariable justify
set m [menu .mb.menu -tearoff 0]
.mb configure -menu $m
$m add radiobutton -value Left -variable justify \
    -label Left
$m add radiobutton -value Center -variable justify
    -label Center
$m add radiobutton -value Right -variable justify \
    -label Right
$m add radiobutton -value Justified -variable justify \
    -label Justified
label .l -text Alignment:
grid .l  -row 0 -column 0 -sticky e
grid .mb -row 0 -column 1 -sticky w
set justify Left
```

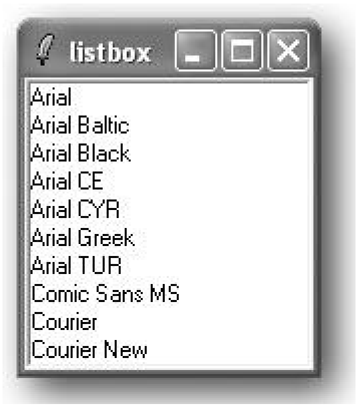
Figure 18.11 Example menubutton as it appears normally (a) and when posted (b)



18.8 Listboxes

A listbox is a widget that displays a collection of strings and allows the user to select one or more of them. For example, the listbox in Figure 18.12 displays the names of the available font families. If a listbox contains too many entries to display at once, as in Figure 18.12, it displays as many as will fit in the window. Scrollbars can be associated with listboxes as described in Section 18.9, and listboxes can be scrolled horizontally if their entries are too wide for their windows.

Figure 18.12 A listbox that displays all font families available in the system



The entries in a listbox can be managed in two ways. One way is through the configure option `-listvariable`. This option takes the name of a global or namespace variable that contains a list of values. As the variable is modified, the listbox updates to display the contents of the list. The `listbox` widget command also provides several actions for manipulating entries, such as `insert` for adding new entries, `delete` for deleting entries, and `get` for retrieving entries. Modifying the variable value using the variety of Tcl list commands is often easier than using the listbox widget commands directly. The script implementing the listbox in Figure 18.12 manipulates the associated list variable to fill the listbox:

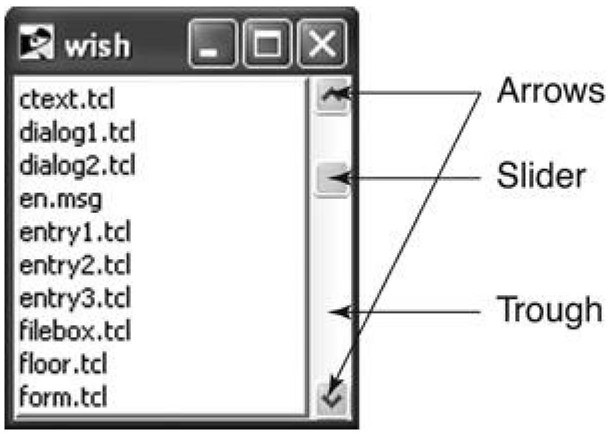
```
listbox .fonts -listvariable ::fontlist
grid .fonts
set ::fontlist \
    [lsort -dictionary [font families]]
bind .fonts <Double-Button-1> {
    %W configure -font [list [%W get [%W curselection]] 12]
}
```

Listboxes are usually configured so that the user can select an entry by clicking on it with mouse button 1. In some cases the user can also select a range of entries by pressing and dragging with button 1. Both single-select and multiple-select modes are supported. (See the reference documentation for more information.) Selected entries appear in a different color or possibly some other style or effect. Once the desired entries have been selected, the user typically uses them by invoking another widget, such as a button or menu. For example, the user might select a font name from the listbox in Figure 18.12 and then click on a button widget to apply that font to some object; the Tcl script associated with the button can read out the selected listbox entry.
It's also common for a double-click on a listbox entry to invoke an operation on the selected entry. For example, the script for Figure 18.12 creates a binding for double-clicks that sets the font of the listbox from the selected font family using the `get` widget command. The script fills the listbox by scanning through the list of font family names, sorted alphabetically, and entering them by storing the list in the listbox's associated list variable. A user can click on an entry like "Courier" to select it. A double-click causes the listbox's font to change to the selected font.

18.9 Scrollbars

Scrollbars control the views in other widgets. Each scrollbar widget is associated with some other widget such as a listbox or entry. For example, Figure 18.13 shows a scrollbar next to a listbox that displays the names of all the files in a directory. The scrollbar displays *arrows*, usually at each end, and a rectangular *slider* in a *trough* in the middle. The size and position of the slider indicate which of the listbox's entries are currently visible in its window. In Figure 18.13 the slider covers the bottom 20% of the area of the trough; this means that the listbox is currently displaying the last 20% of its entries. The user can adjust the view by clicking mouse button 1 on the arrows, which moves the view a small amount in the direction of the arrow, or by clicking in the trough on either side of the slider, which moves the view by one full screen in that direction. The view can also be changed by pressing on the slider and dragging it. The listbox displays the file names in a directory, and the scrollbar is used to change the view in the listbox.

Figure 18.13 A scrollbar and a listbox working together



18.9.1 Scrolling a Single Widget

The scrollbar uses its `-command` option to notify the listbox when the user invokes scrolling operations, and the listbox uses its `-yscrollcommand` option to notify the scrollbar when it changes its view. For example, Figure 18.13 shows a listbox with a scrollbar, the script for which is as follows:

```
listbox .files -relief raised -borderwidth 2 \
    -yscrollcommand ".scroll set" \
    -listvariable files
scrollbar .scroll -command ".files yview"
grid .files  -row 0 -column 0 -sticky nsew
grid .scroll -row 0 -column 1 -sticky ns
grid rowconfigure . 0 -weight 1
grid columnconfigure . 0 -weight 1
set files [lsort -dictionary \
    [glob -directory $tk_library/demos -tails *]]
```

A scrollbar interacts with its associated widget using Tcl scripts. For example, consider the script for Figure 18.13. The listbox is configured with a `-yscrollcommand` option whose value is `.scroll set`. Whenever the view in the listbox changes, the listbox appends two numbers to the `-yscrollcommand` value to generate a command such as

```
.scroll set 0.80 1.0
```

The numbers give the percentage of the whole represented by the slider. The first fraction indicates the first information in the document that is visible in the window, and the second fraction indicates the information just after the last portion that is visible. The preceding command corresponds to the view in Figure 18.13. Then the listbox evaluates the command. In this case the command happens to be the widget command for the scrollbar, and it invokes the `set` action, which expects two arguments in just the form provided by the listbox. The `set` widget command causes the scrollbar to redraw its slider to correspond to the information in the arguments.
When the user clicks on the scrollbar to change the view in the listbox, the scrollbar uses its `-command` option to notify the listbox. For example, suppose that the user clicks on the up arrow in Figure 18.13. The scrollbar generates a Tcl command by appending `scroll -1 unit` to the value of its `-command` option:

```
.files yview scroll -1 unit
```

Then the scrollbar evaluates the command. The widget command for the listbox has an action that takes exactly this form, and it causes the listbox to adjust its view so that the entry prior to the current top entry is displayed on the top line. After adjusting its view, the listbox invokes its `-yscrollcommand` option to notify the scrollbar of the new view, and the scrollbar then redraws its slider. It is up to each widget to define what constitutes a “unit”; in this case, the listbox defines a unit as one entry. When the slider is moved, a different command is generated. In this case the command directs the widget to move to a new fraction offset; for example:

```
.files yview moveto 0.73
```

Here, the widget is directed to scroll so that the information at the top of the display is 73% from the beginning of the document.
This same approach works for all widgets that support scrolling, and it could support different implementations of scrollbars, too. All that a widget must do to be scrollable is to provide a `yview` action for its widget command and a `-yscrollcommand` option. Scrollbars are not hard-wired to work with any

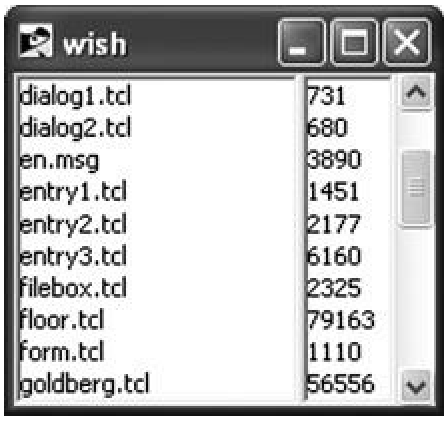
particular widget or widget class. Instead, the information is provided by setting the scrollbar's `-command` option. Likewise, it is possible to build new kinds of scrollbar widgets and have them work with any of the existing widgets as long as they provide a `-command` option and a `set` action for their widget commands.
Tk supports horizontal scrolling as well as vertical scrolling. Scrollbars have a vertical orientation by default, but they can be oriented horizontally by specifying the option `-orient horizontal`. Listboxes have a separate `-xscrollcommand` option and an `xview` action for their widget commands to support horizontal scrolling.

18.9.2 Synchronized Scrolling of Multiple Widgets

It is not necessary to connect the scrollbar and widget directly. A procedure can be used in between to do things like scrolling multiple widgets with a single scrollbar. The trick to getting accurate synchronized scrolling is to identify a master widget that will control the scrollbar and the slave widgets. The accuracy comes into play when there are slight rounding errors in the floating-point calculations performed by each widget. When these calculations are forced to one widget, the rest stay in sync. In addition, a number of Tk extensions provide for multicolumn scrolling widgets to deal with the same problem.
The following script creates what appears to the user as a two-column scrolled list, as shown in Figure 18.14:

```
# called by the listbox
proc Yset {widgets master sb args} {
    if {$master eq "master"} {
        # Only the master sets the scrollbar
        $sb set {expand}$args
        set wl [lrange $widgets 1 end]
    } else {
        set wl [lrange $widgets 0 0]
    }

    Yview $wl moveto [lindex $args 0]
}
# called by the scrollbar
proc yview {widgets args} {
    foreach w $widgets {
        $w yview {expand}$args
    }
}
set widgets [list .lb1 .lb2]
.lb1 configure \
    -yscrollcommand [list yset $widgets master .sb]
.lb2 configure \
    -yscrollcommand [list yset $widgets slave .sb]
.sb configure -command [list yview $widgets]
```

Figure 18.14 Scrolling multiple widgets together



```
dialog1.tcl     731
dialog2.tcl     680
en.msg          3890
entry1.tcl      1451
entry2.tcl      2177
entry3.tcl      6160
filebox.tcl     2325
floor.tcl       79163
form.tcl        1110
goldberg.tcl    56556
```

18.10 Scales

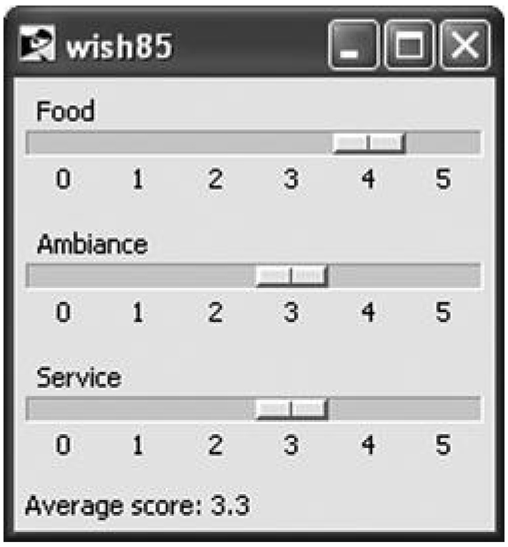
A scale widget has an appearance similar to that of a scrollbar, but instead of scrolling another widget, it is used to select a numerical value by moving the slider along the trough. You specify the minimum and maximum values for the scale with its `-from` and `-to` options. Moving the slider changes the value in increments specified by `-resolution`, which defaults to `1`. The `-orient` option determines whether the scale is `vertical` or `horizontal`, and you can control the size of the scale with the `-length` option to specify the long dimension and the `-width` option to specify the narrow dimension of the trough, both of which accept any screen distance. By default, the scale displays its current value, but you can disable that by setting `-showvalue` to false. You can display an optional text label by providing a string value to the `-label` option. You can also request the display of numerical tick marks by providing the numerical interval to the `-tickinterval` option.

You can use the `-variable` option to link the scale to a global or fully qualified namespace variable so that the value of the scale and the value of the variable are automatically synchronized. You can also retrieve the value of the scale with the `get` subcommand or set it with the `set` subcommand. The scale also supports a `-command` option, which lets you register a command to execute whenever the value changes; the scale automatically appends the updated value of the scale as an argument to the command before invoking it.

The following script creates three horizontal scale widgets, each allowing integer values ranging from 0 to 5. Each scale has a descriptive text label and tick marks, but the value is not displayed. Any time a scale changes its value, it calls the `updateScore` procedure to update the average value of the scales, which is displayed through a label widget. Figure 18.15 shows the result of running the script.

```
set score "Average score: 0.0"
set food 0
set ambiance 0
set service 0

# Calculate an average of the ratings to display
proc updateScore {val} {
    global food ambiance service score
    set average [expr {($food + $ambiance + $service) / 3.0}]
    set score [format {Average score: %3.1f} $average]
}

scale .food -label "Food" -variable food \
    -length 5c -width .25c -from 0 -to 5 \
    -resolution 1 -tickinterval 1 -showvalue 0 \
    -orient horizontal -command {updateScore}
scale .ambiance -label "Ambiance" -variable ambiance \
    -length 5c -width .25c -from 0 -to 5 \
    -resolution 1 -tickinterval 1 -showvalue 0 \
    -orient horizontal -command {updateScore}
scale .service -label "Service" -variable service \
    -length 5c -width .25c -from 0 -to 5 \
    -resolution 1 -tickinterval 1 -showvalue 0 \
    -orient horizontal -command {updateScore}
label .score -textvariable score

grid .food -padx 2 -pady 2 -sticky w
grid .ambiance -padx 2 -pady 2 -sticky w
grid .service -padx 2 -pady 2 -sticky w
grid .score -padx 2 -pady 2 -sticky w
```

Figure 18.15 Examples of scale widgets



Average score: 3.3

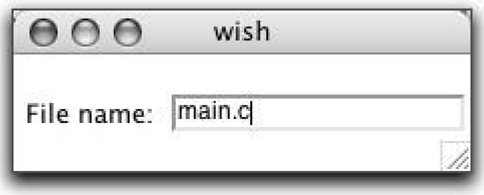
18.11 Entries

An entry is a widget that allows the user to type in and edit a one-line text string. Tk provides two different kinds of entry widgets: entry and spinbox. The spinbox combines an entry with up/down buttons.

18.11.1 Entry Widget

Figure 18.16 shows an entry widget that might be used for entering a file name. To enter text into an entry widget, the user clicks mouse button 1 in the entry. This makes a blinking vertical bar appear, called the *insertion cursor*. The user can then type characters, which are inserted at the point of the insertion cursor. The insertion cursor can be moved by clicking anywhere in the entry's text. Text in an entry can be selected by pressing and dragging with mouse button 1, and it can be edited with a variety of keyboard actions, such as Delete or Backspace to delete the character just before the insertion cursor; see the reference documentation for details. The widget command for entries provides actions such as `insert` for inserting text, `delete` for deleting text, `icursor` for positioning the insertion cursor, and `index` for finding the character displayed at a particular position in the window.

Figure 18.16 An entry widget with an identifying label



Here is the script for the widget shown in Figure 18.16:

```
label .label -text "File name:"
entry .entry -width 20 -relief sunken -bd 2 \
    -textvariable name
grid .label .entry -padx 1m -pady 2m
```

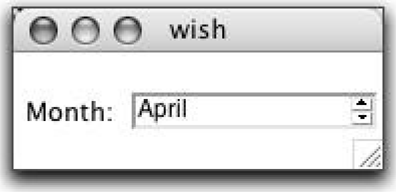
The code specifies a `-width` option for the entry, which indicates how wide the window should be in characters. If the text becomes too long to fit in this amount of space, only a portion of it is displayed; the view can be adjusted with an associated scrollbar widget, the arrow keys, the Home key, or the End key. The script for Figure 18.16 also specifies a `-textvariable` option, which ties the text in the entry to the contents of the global variable `name`. Whenever the text in the entry is modified, `name` is updated and vice versa.

18.11.2 Spinbox

A spinbox provides all the features of an entry widget plus a pair of up/down buttons. Pressing one of the buttons moves or *spins* through a fixed set of ascending or descending values such as times, dates, or integers. The value may be edited directly as in an entry widget unless the `readonly` state is set. To spin through a list of months, for example, the `-values` option is used, as shown in Figure 18.17. In this example the entry box is not editable:

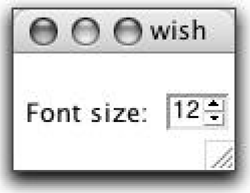
```
set months {January February
  March April May June July
  August September October
  November December}
label .label -text "Month:"
spinbox .spin -width 2 \
    -relief sunken -bd 2 \
    -textvariable month \
    -state readonly \
    -values $months
grid .label .spin -padx 1m \
    -pady 2m
```

Figure 18.17 A spinbox allowing selections from a list of values



A spinbox also can be set to spin through a range of numerical values with the `-from`, `-to`, and `-increment` options. Figure 18.18 shows an example of a spinbox that can spin through integer values from 6 to 72 in increments of 2. Because the `-state` is set to `normal`, the font size also may be set by typing in a number directly:

Figure 18.18 A spinbox that spins through a range of integer values



Font size: 12

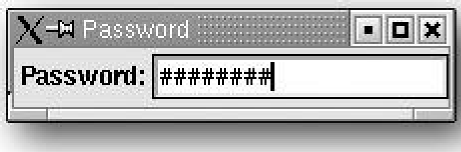
```
label .label -text "Font size:"
spinbox .spin -width 2 \
    -relief sunken -bd 2 \
    -textvariable size \
    -from 6 -to 72 \
    -increment 2 \
    -state normal
grid .label .spin -padx 1m \
    -pady 2m
```

18.11.3 The `show` Option

The `-show` option on an entry widget causes the entry to hide the typed-in characters by displaying the `-show` character instead. This is useful for accepting secret information such as passwords without displaying them on the screen. Figure 18.19 demonstrates an entry with the `-show` option created with the following code:

```
label .label -text Password:
entry .passwd -show #
grid .label .passwd -sticky ew
```

Figure 18.19 Example entry widget with the `-show` option used to hide the contents



Password: ########

18.11.4 Validation

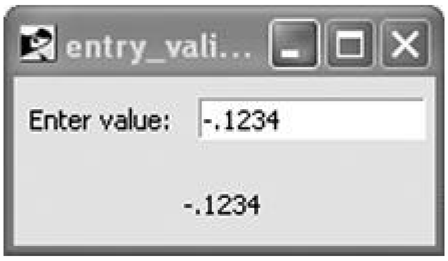
Entry widgets also provide for entry validation through a supplied script. Using the `-validate` option, you can specify that the widget perform validation per keystroke, on a change in focus, or both. Validation is done by a script specified with the `-validationcommand` option. The script must return a Boolean value indicating whether the change is valid and accepted or invalid and rejected. You can also specify an `-invalidcommand` option that provides a script to execute when the validation script returns `0`; a typical use is to invoke `bell`, which rings the display's bell.

Note

An uncaught error during the evaluation of the validation script disables further validation on the entry by automatically setting `-validate` to `none`.

The example in Figure 18.20 validates the entry to allow legal decimal integer or floating-point values. Notice in the code for this example that the `-validatecommand` option has an argument that begins with `%`. Such arguments are substituted with appropriate values before evaluation by the entry widget. In the case of an entry widget, `%W` is replaced with the name of the widget, `%P` with the value of the entry if the edit is allowed, and `%S` with the text string being inserted or deleted. This is a convenient way to pass information to the command about the state of the widget. There are several places throughout Tk where percent substitutions are performed. See the entry reference documentation for a complete list of percent substitutions that are supported for validation.

Figure 18.20 Entry validation



This example script validates the entry by using the Tcl `string is` command to check if the value can be represented as a double-precision-floating point value. But because the script is called for each keystroke, it must also accept some intermediate strings as valid as well, such as if the user has entered an optional sign character but not yet entered any digits. It does this by calling `regexp` if the `string is` command returns `0`. Note that `string is` returns `1` if its value is an empty string, so this also allows a user to delete all of the characters from the entry. The script also "echoes" the value entered in a label beneath the entry:

```
label .label -text "Enter value:"
entry .value -width 15 -validate all \
    -validatecommand { CheckValue %P }
label .echo -textvariable echo
grid .label .value -padx 1m -pady 2m
grid .echo  -      -padx 1m -pady 2m

proc CheckValue { newValue } {
  global echo
  if { [string is double $newValue]
        || [regexp -- {^[+-]?\.?$} $newValue]}
      set echo $newValue
      return 1
  } else {
      return 0
  }
}
```

Note

Validating an entry is very complicated since the value in the entry box is "invalid" most of the time. These invalid intermediate points must be allowed or else you will never get to a valid entry. For this reason, validating on keystrokes is rarely useful. This example illustrates both the capability of the entry box to perform validation and the complexity of doing so.

18.12 Menus

Tk's menu widget is a building block that can be used to implement several varieties of menus, such as pull-down menus, cascading menus, and pop-up menus. A menu is a toplevel widget that contains a collection of *entries* arranged in a column, as shown in Figure 18.21. Menu entries are not distinct widgets, but they behave much like buttons, checkbuttons, and radiobuttons. The following types of entries may be used in menus:

- `command`

Similar to a button widget. Displays a string or image and invokes a Tcl script when mouse button 1 is released over it.

- `checkbutton`

Similar to a checkbutton widget. Displays a string or bitmap and toggles a variable between `0` and `1` when button 1 is released over the entry. Also displays a check mark selector like a checkbutton widget.

- `radiobutton`

Similar to a radiobutton widget. Displays a string or image and sets a variable to a particular associated value when button 1 is released over it. Also displays a radio selector in the same way as a radiobutton widget.

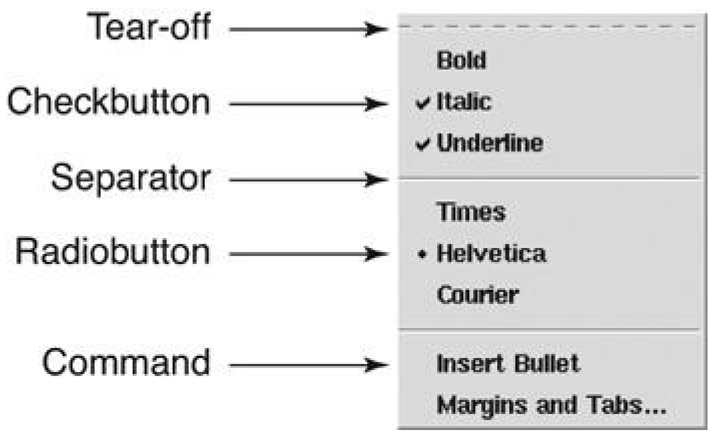
- `cascade`

Similar to a menubutton widget. Posts a cascaded submenu when the mouse passes over it. See below for more details.

- `separator`

Displays a horizontal line for decoration. Does not respond to the mouse.

Figure 18.21 A menu



Menu entries are created with the `add` widget command for the menu, and they have configuration options that are similar to those for buttons, checkbuttons, and radiobuttons.

Menus appear on the screen only for brief periods of time. They spend most of their time in an invisible state called *unposted*. For a menu to be invoked, it must first be *posted*. This is done with the `post` action for its widget command; for example, the menu in Figure 18.21 could be posted with the command `.m post 100 100`. Once a menu is posted the user can move the pointer over one of the menu's entries and release button 1 to invoke the entry. After the menu has been invoked, it is usually unposted until it is needed again.

The following script creates a menu. The `add` action for the menu's widget command is invoked to create the individual entries in the menu.

```
option add *Menu.tearOff 0
menu .m
.m add checkbutton -label Bold -variable bold
.m add checkbutton -label Italic -variable italic
.m add checkbutton -label Underline -variable underline
.m add separator
.m add radiobutton -label Times -variable font \
    -value Times
.m add radiobutton -label Helvetica -variable font \
    -value Helvetica
.m add radiobutton -label Courier -variable font \
    -value Courier
.m add separator
.m add command -label "Insert Bullet" \
    -command "insertBullet"
.m add command -label "Margins and Tabs..." \
    -command "mkMarginPanel"
```

Menus can be posted and unposted in various ways to achieve different effects. The next sections describe pull-down menus and cascaded menus, where posting and unposting are handled automatically. Other approaches such as pop-up menus and option menus are also possible.

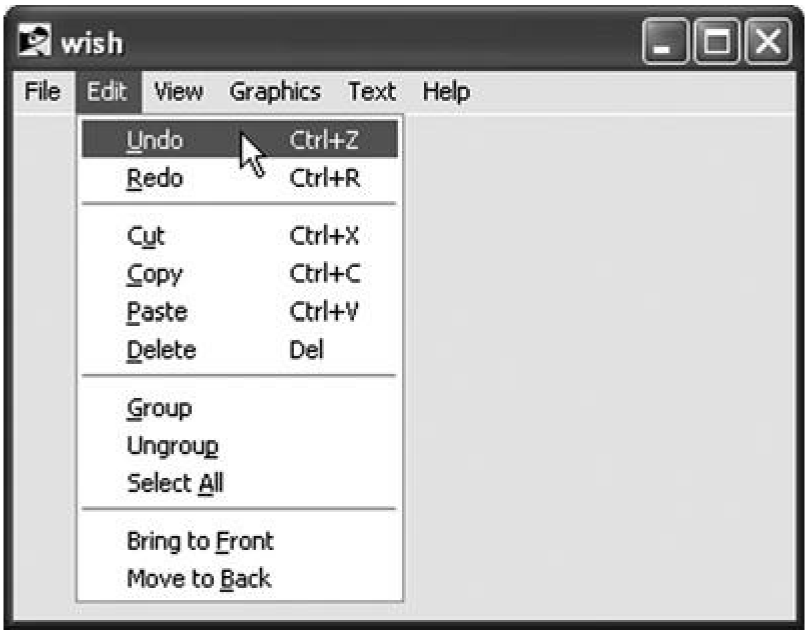
Note

Tk menus support a "tear-off" feature. If enabled, a special separator appears as the first option; if selected, the menu is displayed as a separate toplevel window. Tear-off menus were a typical feature of Motif interfaces, and so they are enabled on Tk menus by default. Tear-offs are rarely found in modern interfaces, though, so typically you'll want to disable the tear-off feature by default. This is accomplished by including the command `option add *Menu.tearOff 0` to your script before creating any menu widgets. This works by setting a value that applies to all menus in the option database, described in Chapter 28.

18.12.1 Pull-Down Menus

Menus are most commonly used in a *pull-down* style. In this style the application displays a *menu bar* near the top of its main window, as shown in Figure 18.22. A menu bar is drawn at the top of the window and contains several entries that act like buttons, each of which is associated with a menu. (On Mac OS X, the menu bar appears at the top of the screen.) When a user presses mouse button 1 over a menu entry, the associated menu is posted underneath the entry (for example, the Edit menu is posted in Figure 18.22). Then the user slides the pointer down over the menu, with the mouse button still pressed, and releases the mouse button over the desired entry. When the button is released, the menu entry is invoked and the menu is unposted. The user can release the mouse button outside the menu to unpost it without invoking an entry.

Figure 18.22 An example of pull-down menus with one menu posted



The following script creates a menu bar. The menu bar frame (`.mbar`) has six children, each of which is a cascaded menu with an associated menu. The definition of the menus is incomplete: only part of the definition for a single menu is shown.

```
option add *Menu.tearOff 0
menu .mbar
. configure -menu .mbar

.mbar add cascade -label File -menu .mbar.file -underline 0
.mbar add cascade -label Edit -menu .mbar.edit -underline 0
.mbar add cascade -label View -menu .mbar.view -underline 0
.mbar add cascade -label Graphics -menu .mbar.graphics \
    -underline 0
.mbar add cascade -label Text -menu .mbar.text -underline 0
.mbar add cascade -label Help -menu .mbar.help -underline 0

menu .mbar.text

.mbar.text add checkbutton -label Bold -variable bold \
    -underline 0
.mbar.text add checkbutton -label Italic -variable italic \
    -underline 0
...
```

If the user presses button 1 over a menu entry and then moves the pointer over another menu entry, the old entry unposts its menu and the new entry posts its menu. This allows the user to scan all of the menus by sliding the pointer across the menu bar.

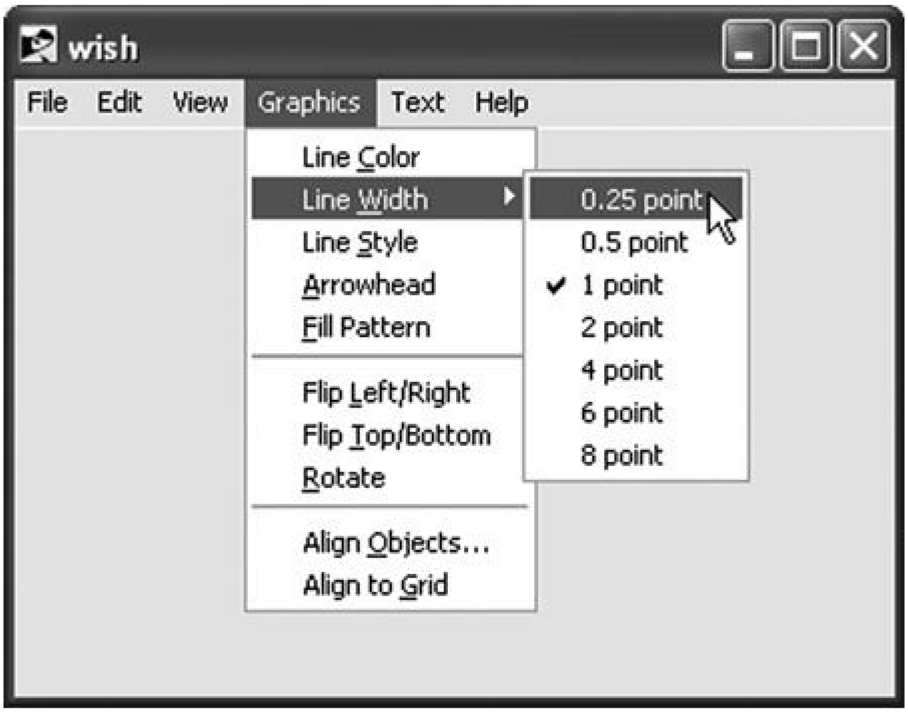
If the user releases the mouse button over an entry, the menu stays posted and the user is not able to do anything else with the application until the menu is unposted, which the user does by invoking one of its entries, or clicking outside the menu without invoking any entry. Situations like this where a user must respond to a particular part of an application and cannot do anything with the rest of the application until responding are called *modal* user interface elements. Menus and dialog boxes are examples of modal interface elements, which are implemented using the grab mechanism described in Chapter 27.

In Tk, menu bars are an integral part of a toplevel window. The toplevel manages the menu bar placement differently for different platform environments. A menu bar is created by building a menu and then giving it to the toplevel for display by configuring the `-menu` option on the toplevel window. In the script for Figure 18.22, the `-menu` and `-underline` options are specified for each entry in addition to its label string. The `-menu` option identifies the menu associated with the entry. The `-underline` option gives the index of a character to underline when displaying the label in the entry. This character is used for keyboard traversal, as described in Section 18.12.3.

18.12.2 Cascaded Menus

A *cascaded menu* is a menu that is subservient to another menu. It is associated with a cascade menu entry in its parent menu. Note that the menu bar is simply a list of cascaded menus. When the pointer passes over a cascade entry in a menu, the associated menu is posted just to the right of the cascade entry, as shown in Figure 18.23. The user can then slide the pointer to the right onto the cascaded menu and invoke an entry in it. After an entry has been invoked in a cascaded menu, both it and its parent are unposted. All that is needed to create a cascaded menu is to define the cascaded menu and create a cascade entry in its parent, using the `-menu` option in the cascade entry to specify the name of the cascaded menu. Menus can be cascaded to any depth, but for best usability it is a good rule of thumb to keep the maximum depth to three.

Figure 18.23 A cascaded menu



The following example shows the changes to the script for Figure 18.22 that are needed to create a cascaded menu:

```
menu .mbar.graphics
...
.mbar.graphics add cascade -label "Line Color" \
    -menu .mbar.graphics.color -underline 5
.mbar.graphics add cascade -label "Line Width" \
    -menu .mbar.graphics.width -underline 5
...
.mbar.graphics.width add radiobutton -label "0.25 point" \
    -variable lineWidth -value 0.25
...
```

18.12.3 Keyboard Traversal and Shortcuts

Pull-down menus can be posted and invoked without the mouse using a technique called *keyboard traversal*. One of the letters in each menu entry is selected as the traversal character for that entry with the `-underline` option; it is underlined in the entry's window. If that letter is typed while the Alt key is held down, the entry's menu is posted. Or the user can press the F10 key to post the leftmost menu in the menu bar. Once a menu has been posted, the arrow keys can be used to move among the menus and their entries. The left and right arrow keys move left or right among the toplevel entries, unposting the menu for the previous entry and posting the menu for the new one. The up and down arrow keys move among the entries in a menu, activating the next higher or lower entry. You can press the Return key to invoke the active menu entry, or Escape to abort the menu traversal without invoking anything. Traversal characters can also be defined for individual menu entries with the `-underline` option, as shown in the code for Figure 18.22. The traversal character is underlined when the menu entry is displayed, and if it or its lowercase equivalent is typed at a time when the menu is posted, the entry is invoked.

In many cases it is possible to invoke the function of a menu entry without even posting the menu by typing *keyboard shortcuts*. If there is a shortcut for a menu entry, the keystroke for the shortcut is displayed at the right side of the menu entry (for example, you might want to display `Ctrl+Z` as the shortcut for an Undo menu entry). This key combination may be typed in the application to invoke the same function as the menu entry (for example, type `z` while holding the Control key down to invoke the Undo operation without going through the menu). The `-accelerator` option for a menu entry specifies a string to display at the right side of the entry. Common modifiers, which may be combined, include `Control`, `Ctrl`, `Shift`, `Command`, `Cmd`, `Option`, and `Opt`.

Note

On Mac OS X, these modifiers are mapped automatically to the corresponding modifier icons that appear in menus.

A key binding must also be defined in order for the shortcut to work. This is done using the `bind` command like so:

```
bind . <Control-Key-z> {.mbar.edit invoke "Undo"}
```

When the keyboard event is bound to ., the toplevel window, the event is caught no matter which internal window has the keyboard focus. More information about the `bind` command and event propagation can be found in Chapter 22.

18.12.4 Platform-Specific Menus

Each platform has one or more menus for which Tk provides special handling. To take advantage of this feature, you must give particular names to the menu widgets you create in support of these special menus. In all cases, these menu widgets must be created with the name of the menu bar menu concatenated with the special name (that is, *`.menuBar. special`* for your application's main window, or *`.toplevel.menuBar.special`* for a secondary toplevel).
On X11 windowing systems, the Help menu should always appear at the far right of the window. Tk does this automatically as long as you use `.help` as the last component of the name of the menu widget implementing the Help menu (for example, `.mbar.help` for your application's main window, or `.doc1.mbar.help` for a secondary toplevel).
On Windows systems, you can access the system menu for a window, which is the menu posted by clicking on the application's icon in the window title bar. Any entries you add appear after the standard Windows entries. To access the system menu, use `.system` as the last component of the menu's name (for example, `.mbar.system`).
On Mac OS X, you can append entries to the standard Help menu by creating a menu widget whose name ends in `.help` (for example, `.mbar.help`). Mac OS X also has what's known as an application menu, which is the second menu in the menu bar, the title of which is the same as your

application name. To access the application menu, create a menu whose name ends in `.apple` (for example, `.mbar.apple`). Any entries that you add to the `.apple` menu appear before the system-standard entries.

Note

The `.apple` menu really does access the application menu (second from the left), not the leftmost menu with the apple icon. The name is a holdover from pre–OS X days when application-specific entries really did go under the apple (icon) menu.

The Mac OS X application menu also provides a standard "Preferences" item. This item is disabled unless you define a procedure named `::tk::mac::ShowPreferences`. Typically, you would define this procedure to display any preferences dialog that you might have implemented.

18.12.5 Pop-up Menus

The `tk_popup` utility is a convenience procedure that makes it easy to post pop-up menus. Pop-up menus are typically bound to mouse button 3 on Windows and Unix systems, corresponding to the right mouse button. On Mac OS X, they should typically respond to a click of the left (or only) button while the Control key is held, or the right button on a multibutton mouse, which maps to mouse button 2. Therefore, in a platform-independent application, you should use the `tk windowingsystem` command to determine if your application is running on the native Mac OS X windowing system, which is indicated by a return value of `aqua`.
Pop-up menus also are typically sensitive to the location of the mouse pointer at the time the button is pressed. This means that the menu that appears has actions related to the item under the mouse pointer. The `tk_popup` utility makes it easy to bind the mouse button to a menu:

```
if {[tk windowingsystem]=="aqua"} {
    bind .lbox <ButtonPress-2> {tk_popup .menu %X %Y}
    bind .lbox <Control-ButtonPress-1> {tk_popup .menu %X %Y}
} else {
    bind .lbox <ButtonPress-3> {tk_popup .popup_menu %X %Y}
}
```

When the appropriate mouse button is pressed over the `.lbox` widget, the `%X` and `%Y` arguments are replaced with the x- and y-coordinates of the mouse pointer. In this example, the `tk_popup` utility posts `.popup_menu` at the appropriate location relative to the mouse pointer location, usually immediately below and to the right of the pointer. The `bind` command is discussed in more detail in Chapter 22.

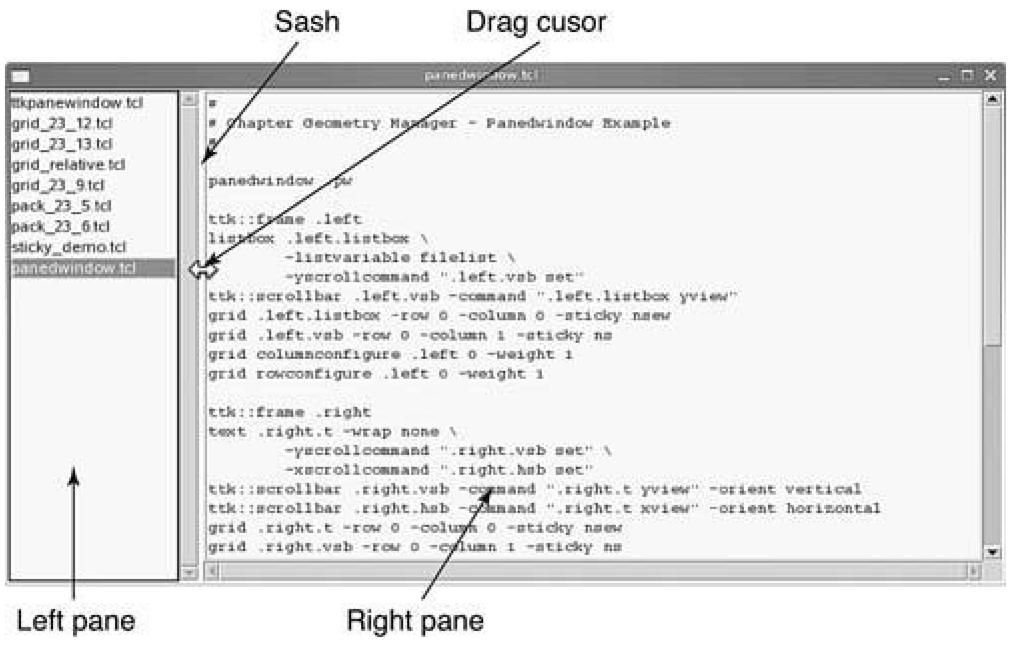
Note

The platform dependency of the bindings could also be handled through the definition of virtual events, which are also described in Chapter 22.

18.13 Panedwindow

Several commonly used applications, like e-mail clients and file browsers, have a window layout similar to that shown in Figure 18.24. The main window is divided into two or more panes with a movable divider between them. This is handled in Tk with the panedwindow widget. Panes can be arranged either horizontally or vertically, depending on whether you set the panedwindow's `-orient` option to `horizontal` or `vertical`, respectively. You can set the `-showhandle` option to a Boolean value to specify whether or not you want to display a resize handle on each sash separating panes.

Figure 18.24 The panedwindow manages a list of widgets into a horizontal row or vertical column.



```
#
# Chapter Geometry Manager - Panedwindow Example
#

panedwindow .pw

ttk::frame .left
listbox .left.listbox \
        -listvariable filelist \
        -yscrollcommand ".left.vsb set"
ttk::scrollbar .left.vsb -command ".left.listbox yview"
grid .left.listbox -row 0 -column 0 -sticky nsew
grid .left.vsb -row 0 -column 1 -sticky ns
grid columnconfigure .left 0 -weight 1
grid rowconfigure .left 0 -weight 1

ttk::frame .right
text .right.t -wrap none \
        -yscrollcommand ".right.vsb set" \
        -xscrollcommand ".right.hsb set"
ttk::scrollbar .right.vsb -command ".right.t yview" -orient vertical
ttk::scrollbar .right.hsb -command ".right.t xview" -orient horizontal
grid .right.t -row 0 -column 0 -sticky nsew
grid .right.vsb -row 0 -column 1 -sticky ns
```

Each pane is a cell that contains another widget. Typically, you arrange the contents of each pane into a frame widget, then add each frame to the panedwindow using the `add` subcommand. (Chapter 21 explains how to arrange widgets within a frame.) In Figure 18.24 the left pane contains a listbox showing a list of files, and the right pane contains a text widget to display the contents of the selected file.
The following script shows how to add existing widgets to a panedwindow:

```
# The .left frame contains a listbox and scrollbar
# The .right frame, a text widget and scrollbars
# These are placed in a panedwindow:

panedwindow .pw -orient horizontal
.pw add .left .right
```

Each pane that you add is controlled by a set of options. You can set the option values when you add the pane with `add`, or afterward with the `paneconfigure` subcommand. You can query the value of a pane option with the `panecget` subcommand; for example:

```
  .pw panecget .left -sticky
⇒ nesw
  .pw paneconfigure .left -minsize 2i
```

The following pane options are supported:

- `-after` *`widget`*

Inserts the widget after the *`widget`* specified, which should be the name of a widget already managed by the panedwindow.

- `-before` *`widget`*

Inserts the widget before the *`widget`* specified, which should be the name of a widget already managed by the panedwindow.

- `-height` *`size`*

Specifies a height for the widget, expressed as a screen distance. If *`size`* is an empty string, or if `-height` is not specified, the height requested internally by the widget is used initially; the height may later be adjusted by the movement of sashes in the panedwindow.

- `-hide` *`boolean`*

Controls the visibility of a pane. When *`boolean`* is true, the pane is not visible, but it is still maintained in the list of panes.

- `-minsize` *`size`*

Specifies that the size of the pane cannot be made less than *`size`*, expressed as a screen distance.

- `-padx` *`size`*

Specifies horizontal padding added around the widget, expressed as a screen distance.

- `-pady` *`size`*

Specifies vertical padding added around the widget, expressed as a screen distance.

- `-sticky` *`style`*

Controls the position and stretching of a widget if a widget's pane is larger than its requested dimensions. *`style`* is a string that contains zero or more of the characters `n`, `s`, `e`, or `w`. Each letter refers to a side (north, south, east, or west) to which the widget "sticks." If both `n` and `s` (or `e` and `w`) are specified, the window is stretched to fill the entire height (or width) of its pane.

- `-stretch` *`when`*

Controls how extra space is allocated to each of the panes. You can set the pane to `always` or `never` stretch. Alternatively, you can specify that the pane stretch only if it is the `first`, `last`, or `middle` pane.

- `-width` *`size`*

Specifies a width for the window, expressed as a screen distance. If *`size`* is an empty string, or if `-width` is not specified, the width requested internally by the window is used initially; the width may later be adjusted by the movement of sashes in the panedwindow.

Widgets previously added to a panedwindow can be removed from it using the `forget` subcommand. You can also retrieve a list of widgets managed by the panedwindow using the `panes` subcommand; the widgets are returned in the order in which they are displayed:

```
.pw panes
⇒ .left .right
```

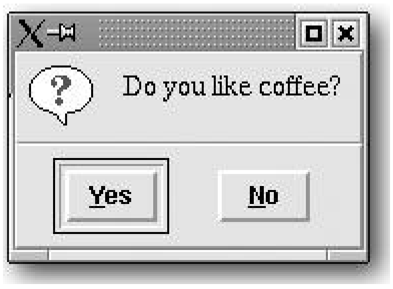
Note

The panedwindow is both a widget and a geometry manager. Geometry management is discussed at length in Chapter 21.

18.14 Standard Dialogs

Tk provides a set of utility procedures and dialogs. Three are described here, but refer to the documentation for a complete list. You can also create your own custom dialogs, a topic that is discussed in Chapter 27.
At times in an application it is necessary to report a condition or ask a question that needs immediate attention. This can be handled easily in Tk using the `tk_messageBox` dialog. Calling this procedure presents a dialog box containing a message and a set of buttons, depending on the `-type` specified. An image is displayed next to the message text as defined by the `-icon` option. This image is used to help the user identify the importance of the message, so the possible values are `error`, `info`, `question`, and `warning`. The image varies from one platform environment to another, and it is typically standardized across all applications for that environment. Figure 18.25 demonstrates a simple `yesno` message box. The `-parent` option specifies an application widget that the `tk_messageBox` uses for positioning the dialog. The dialog box is centered over the top of the parent widget. The message box dialog is *modal* and blocks further execution of the Tcl application until the user responds by pressing one of the buttons. The return value is a string indicating which button was pressed, such as `yes` or `no`.

Figure 18.25 Example of Tk's built-in `messageBox` dialog

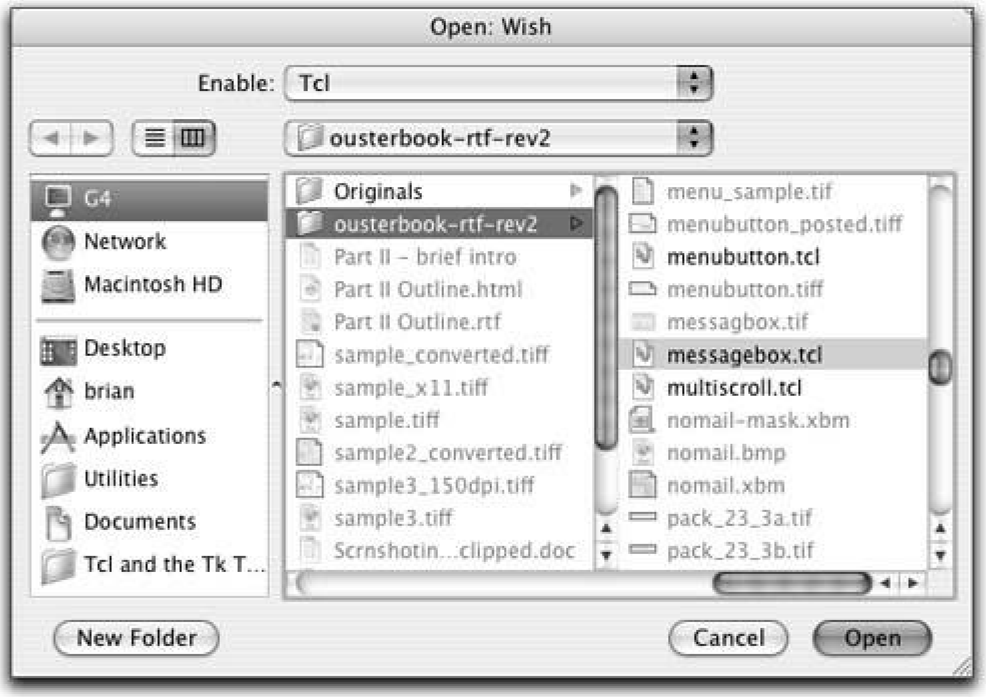


The following script shows a message box and the response. The function returns a value based on which button the user presses.

```
    tk_messageBox -type yesno -icon question \
        -message "Do you like coffee?" -parent .
⇒ yes
```

Selecting operating system files for open and save operations is a common enough task that each platform environment has a standardized way of presenting a dialog for file selection. These dialogs are common across all applications, reducing the learning curve for users. Tk provides access to these common dialogs with the `tk_getOpenFile`, `tk_getSaveFile`, and `tk_chooseDirectory` procedures. When these procedures are called, the Tk application waits until the user completes the file selection. The procedure then returns the selected file or files (or an empty string if the user selected no files), and the Tcl script continues. In the code for Figure 18.26, the `-initialfile` option is used to seed the dialog so that it displays the given file name. The `-filter` can be used to limit the files visible to those that make sense for the application at hand.

Figure 18.26 Invoking the native file navigator dialog boxes



The following command creates a file dialog to open an existing file. The file selected is returned by the call, or an empty string if the dialog box is canceled. A variety of initial conditions (`-initialfile` shown here) and filters can be configured via options to these calls:

```
tk_getOpenFile -initialfile messagebox.tcl
        -filetypes {{Tcl .tcl} {All *}}
```

18.15 Other Common Options

The widget descriptions discussed in this chapter introduced most of the common kinds of options, such as colors and fonts. This section describes a few other options that are supported by many widgets.

18.15.1 Widget State

Interactive widgets and menu entries have a `-state` option associated with them. The default value of `-state` is `normal`, in which case the default widget class bindings make the widget responsive to user interaction. For example, a button responds to a mouse click by invoking its `-command` script, a listbox lets a user scroll through and select its items, an entry widget allows a user

to edit its text, and so forth. Setting the `-state` option to `disabled` causes the default widget class bindings to make the widgets unresponsive to user input; most widgets also change their visual appearance, for example, appearing “grayed out.” Several interactive widgets also support a value of `active`, which is used primarily by the default class bindings to highlight the widget when the mouse pointer is over the widget.
The most common use of the `-state` option in an application is to disable features of an application under certain circumstances and enable them in others. For example, you might enable a Copy button (set its `-state` to `normal`) if the user has selected some text in your application and disable it (set its `-state` to `disabled`) otherwise.
Other widgets have some additional states, such as `readonly` for an entry or spinbox and `hidden` for a canvas; see the appropriate sections of this book and the reference documentation for more information about these states.

18.15.2 Widget Size Options

Many widgets support a `-width` option, and some support a `-height` option, to specify a preferred minimum size. If a widget contains text, these values are usually interpreted in terms of the number of characters wide and the number of lines tall, so the actual size depends on the font and size of the text. If a widget contains an image or bitmap, the values are usually interpreted as pixels. If a widget is not large enough to display all of its contents, it usually clips the contents subject to the `-anchor` option discussed in the next section.
Although you can try to calculate or forecast sizes for your widgets, it’s usually easiest with widgets such as labels and buttons to leave these options with their default value, which is `0`. In this case, the widgets automatically size themselves to be just large enough to accommodate their contents. And in other cases, such as entries or text widgets, you often use the geometry managers to resize the widgets based on the user resizing the window, which overrides any values set for `-height` and `-width`.

18.15.3 Anchor Options

An *anchor position* indicates how to attach one object to another. For example, if the window for a button widget is larger than is needed for the widget’s text, an anchor option may be specified to indicate where the text

should be positioned in the window. Anchor positions are also used for other purposes, such as telling a canvas widget where to position a bitmap relative to a point associated with the item, or telling the packer geometry manager where to position a window in its frame.
Anchor positions are specified using one of the following points of the compass:

- `n`—center of the object's top side
- `ne`—top right corner of the object
- `e`—center of the object's right side
- `se`—lower right corner of the object
- `s`—center of the object's bottom side
- `sw`—lower left corner of the object
- `w`—center of the object's left side
- `nw`—top left corner of the object
- `center`—center of the object

The anchor position specifies the point on the object by which it is to be attached, as if a pushpin were stuck through it at that point, pinning the object someplace. For example, an `-anchor` option of `w` specified for a button means that the button's text or bitmap is to be attached by the center of its left side, and that point is positioned over the corresponding point in the window. Thus, `w` means that the text or bitmap is centered vertically and aligned with the left edge of the window. For bitmap items in canvas widgets, the `-anchor` option indicates where the bitmap should be positioned relative to a point associated with the item; in this case, `w` means that the center of the bitmap's left side should be positioned over the point, so that the bitmap actually lies to the east of the point. Figure 18.8 and Figure 18.9 show the use of the `-anchor` option to cause the buttons to align on the left (for west) side of the window.

18.15.4 Internal Padding

Many widgets support `-padx` and `-pady` options, which control internal padding. The `-padx` option accepts a screen distance specifying the minimum amount of space horizontally between the widget's borders and its content; the `-pady` option determines the minimum vertical spacing between the widget's borders and its content.

18.15.5 Cursor Options

Every widget class in Tk supports a `-cursor` option that determines the image to display for the pointer when it is over that widget. If the `-cursor` option isn't specified or if its value is an empty string, the widget uses its parent's cursor. Otherwise, the value of the cursor option must be a proper Tcl list with one of the following forms:

- `name fgColor bgColor`
- `name fgColor`
- `name`
- `@sourceFile maskFile fgColor bgColor`
- `@sourceFile fgColor`
- `@sourceFile`

In the first three forms `name` refers to one of the predefined cursors. You can find a complete list of all the predefined names in the cursors reference documentation.
If `name` is followed by two additional list elements, as in the command

```
.f config -cursor {arrow red white}
```

the second and third elements give the foreground and background colors to use for the cursor; as with all color values, they may have any of the forms described in Section 18.3. If only one color value is supplied, it gives the foreground color for the cursor and the background is transparent. If no color values are given, black is used for the foreground and white for the background.
If the first character in the `-cursor` value is `@`, the image(s) for the cursor is taken from files in bitmap format rather than the X cursor font. If two file names and two colors are specified for the value, as in the following widget command,

```
.f config -cursor \
        {@cursors/bits cursors/mask red white}
```

the first file is a bitmap that contains the cursor's pattern (`1`s represent foreground and `0`s background) and the second file is a mask bitmap. The cursor is transparent everywhere that the mask bitmap has a `0` value; it displays the foreground or background wherever the mask is `1`. If only one file name and one color are specified, the cursor has a transparent background. This feature is supported only on the X windowing system.
The final form, which works on Windows systems only, loads a Windows system cursor (with either a `.ani` or `.cur` extension) from the file specified by

sourceFile.

On Windows and Macintosh systems, some cursors are mapped to native cursors. The look of these cursors depends on the user's preferences. Some additional cursors are defined that are available only on a Windows or Macintosh platform. See the cursors reference documentation for a complete list.

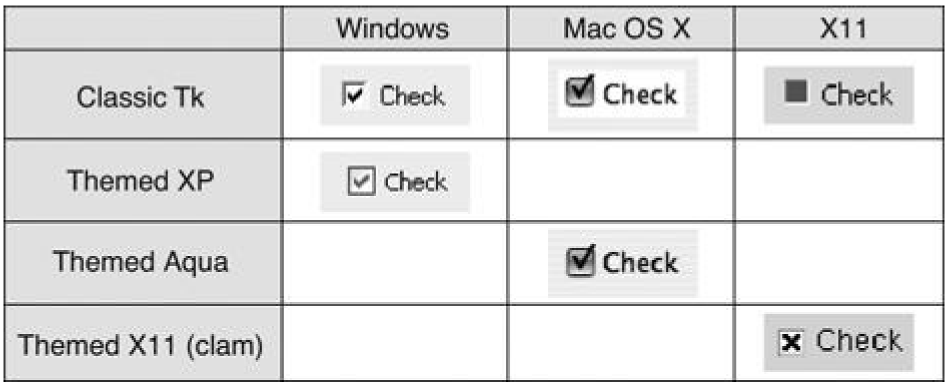
19. Themed Widgets

Themed widgets take a different approach from the classic Tk widget set. They separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance. While classic widgets are configured through the option database or by directly modifying options on each widget, the size, shape, color, fonts, and so on of themed widgets are controlled by their *style*. Themed widgets allow an application's look and feel to be controlled by its environment (e.g., Windows user preferences) or by centralized application-defined styles. The resulting appearance is more consistent throughout the application, as well as with the native look and feel of the user's windowing system.

19.1 Comparing Classic and Themed Widgets

The main advantage to using themed widgets is to give a native look to an application by selecting the *theme* that fits the user's current desktop environment. Classic widgets on Microsoft Windows and Mac OS X try to emulate native widgets for buttons, scrollbars, checkbuttons, and radiobuttons, but these appearances can become dated with newer versions of the windowing system. This is especially apparent with X-based classic widgets, which are still based largely on the look of the Motif widget toolkit. Figure 19.1 illustrates the classic Tk checkbutton widget along with its themed variations for each platform. In some cases the difference may be subtle, but there are small changes that make a big difference, like the background blending on the Mac OS X checkbutton.

Figure 19.1 Classic and themed checkbutton variations



	Windows	Mac OS X	X11
Classic Tk	☑ Check	☑ Check	■ Check
Themed XP	☑ Check		
Themed Aqua		☑ Check	
Themed X11 (clam)			☒ Check

Another benefit to themed widgets is that they are simpler to use than classic widgets. Themed widgets require much less configuration to give your application a consistent and attractive appearance, especially if you want subsets of widgets to have special appearances or behaviors. Customizing classic widgets throughout your application requires you to either set the configuration options for each widget individually or through the Tk option database (described in Chapter 28). For themed widgets, you can define a new style—usually by inheriting the definitions of an existing style—customize specific configuration options for the new style, and then assign the new style to individual widgets as needed. This process is covered in Section 19.9.

Table 19.1 lists the classic widgets and how they compare to the themed widgets. Some of these widgets are the same as the basic Tk widget set, and others provide additional behaviors not found in the classic Tk widgets. Some classic widgets, such as the `canvas` and `text` classes, don’t have a themed equivalent, and the themed widgets include some new widget types, such as the `combobox`, `notebook`, and `treeview` classes.

Table 19.1 Classic Widget Classes and Their Corresponding Themed Widget Classes

Classic widget	Themed widget
`button`	`ttk::button`
`canvas`	
`checkbutton`	`ttk::checkbutton`
	`ttk::combobox`
`entry`	`ttk::entry`
`frame`	`ttk::frame`
`label`	`ttk::label`
`labelframe`	`ttk::labelframe`
`listbox`	
`menu`	
`menubutton`	`ttk::menubutton`
	`ttk::notebook`
`panedwindow`	`ttk::panedwindow`
	`ttk::progressbar`
`radiobutton`	`ttk::radiobutton`
`scale`	`ttk::scale`
`scrollbar`	`ttk::scrollbar`
	`ttk::separator`
	`ttk::sizegrip`
`spinbox`	
`text`	
`toplevel`	
	`ttk::treeview`

Widgets that have both a classic Tk class and a themed widget class are functionally equivalent but not directly interchangeable. The themed widgets do not have options that control color, borders, and so on; instead, these aspects of the widget are controlled by the theme. Also, some themed widgets have different widget commands from their classic counterparts. Subsequent sections of this chapter focus on the new widget classes added by the themed widgets.

Note

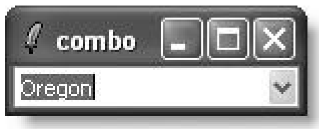
If you decide to replace classic widgets in an existing Tk application with themed widgets to update the application's appearance, you should read the reference documentation carefully to identify the exact differences between the themed widgets and the classic widgets you are replacing.

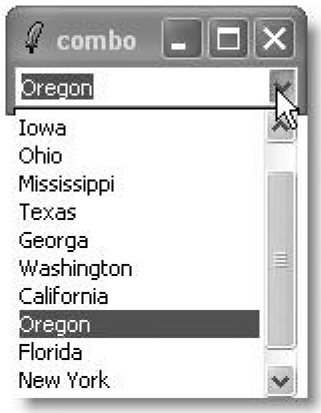
19.2 Combobox

The ttk::combobox widget combines an entry widget with a drop-down listbox. This widget can be used like the spinbox to select from a fixed set of values, but it allows the direct selection of a value without incrementing through the list. Or it can be used to accumulate a history of values that have been entered directly into the entry box. Figure 19.2 shows a combobox used for performing searches. In this example, each time a new search string is typed in, the string is added to the list of values for the combobox. Pressing the down arrow to the right of the entry field displays a drop-down box showing the past search values. Selecting one of these values updates the entry with the new value. The following script creates the combobox shown in Figure 19.2:

```
ttk::combobox .cb
grid .cb
proc UpdateCombo {w} {
    set new [$w get]
    set values [$w cget -values]
    if {$new ni $values} {
        set values [linsert $values 0 $new]
        $w configure -values $values
    }
}

bind .cb <Return> {UpdateCombo %W}
```

Figure 19.2 Example of a themed combobox widget

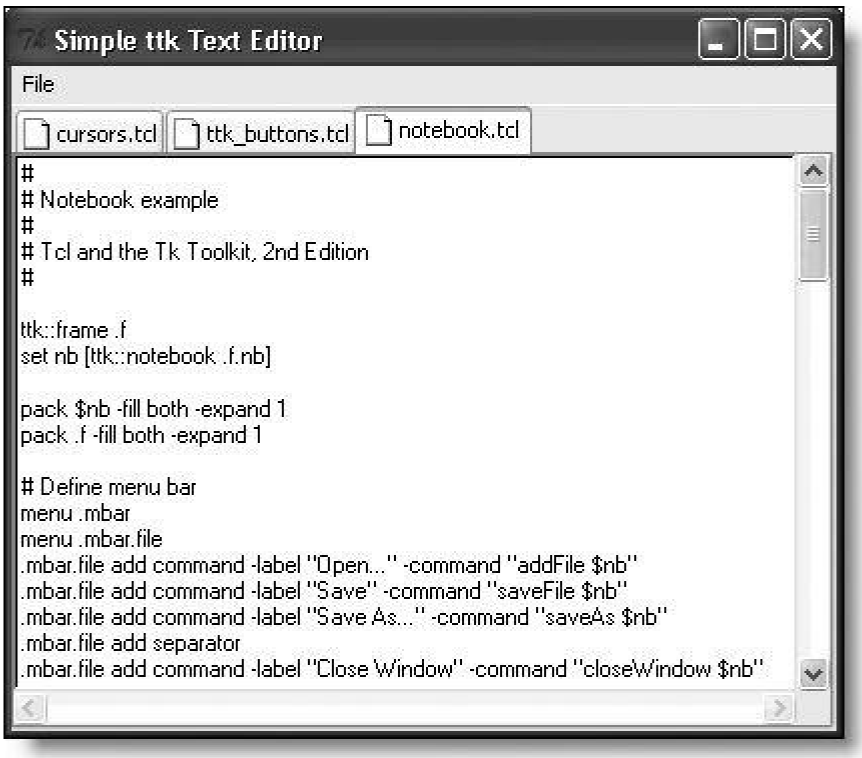




19.3 Notebook

The ttk::notebook widget provides a way to manage windows by dividing them into pages, tying each page to a user-selectable tab, and displaying only one page at a time. This widget is commonly used in complex dialog boxes, and there are other useful applications for it, such as the simple text editor shown in Figure 19.3.

Figure 19.3 Example of a themed notebook widget



```
#
# Notebook example
#
# Tcl and the Tk Toolkit, 2nd Edition
#

ttk::frame .f
set nb [ttk::notebook .f.nb]

pack $nb -fill both -expand 1
pack .f -fill both -expand 1

# Define menu bar
menu .mbar
menu .mbar.file
.mbar.file add command -label "Open..." -command "addFile $nb"
.mbar.file add command -label "Save" -command "saveFile $nb"
.mbar.file add command -label "Save As..." -command "saveAs $nb"
.mbar.file add separator
.mbar.file add command -label "Close Window" -command "closeWindow $nb"
```

A notebook is somewhat similar to a panedwindow in that it acts as a manager for other widgets. When a tab is selected (either programmatically or by a user clicking on it), the notebook displays a window, hiding any child window that was previously displayed. Typically, you arrange the contents of each pane into a ttk::frame or frame widget, then add each frame to the notebook using its `add` subcommand. (Chapter 21 explains how to arrange widgets within a frame.)
Tabs are referenced by an identifier. There are several methods of specifying a tab identifier:

- An integer index starting with `0` for the first tab
- The name of the child widget managed by the notebook
- A screen position of the form `@`*`x, y`*, where *`x`* and *`y`* are expressed as pixels relative to the upper-left corner of the notebook widget
- The literal string `current`, which refers to the currently selected tab

Each child window that you add is controlled by a set of options. You can set the option values when you add the window with `add`, or afterward with the `tab` subcommand, which accepts a tab identifier as its first argument:

notebook tab *tabid ?option? ?value option value ...?*

The following tab options are supported:

- `-compound` *`style`*

Specifies how to display the image relative to the text, when both `-text` and `-image` are present. See Section 19.10 for legal values.

- `-image` *`image`*

Specifies the name of an image object to display in the tab.

- `-padding` *`space`*

Specifies the amount of extra space to add around the outside of the tab window, expressed as a screen distance. The padding is a list of up to four length specifications: *`left`*, *`top`*, *`right`*, and *`bottom`*. If fewer than four elements are specified, *`bottom`* defaults to *`top`*, *`right`* defaults to *`left`*, and *`top`* defaults to *`left`*.

- `-state` *`state`*

Indicates the state, which is either `normal`, `disabled`, or `hidden`. If `disabled`, the tab is not selectable. If `hidden`, the tab is not shown.

- `-sticky` *`style`*

Controls the position and stretching of a widget if a widget's tab window is larger than its requested dimensions. *`style`* is a string that contains zero or more of the characters `n`, `s`, `e`, or `w`. Each letter refers to a side (north, south, east, or west) to which the widget "sticks." If both `n` and `s` (or `e` and `w`) are specified, the window is stretched to fill the entire height (or width) of its window.

- `-text` *`string`*

Specifies a string to be displayed in the tab.

- `-underline` *`index`*

Specifies the integer index (zero-based) of a character to underline in the text string. The underlined character is used for mnemonic activation if `ttk::notebook::enableTraversal` is called.

The `insert` widget command inserts a tab at a specified position:

```
notebook insert position widget ?option? \
    ?value option value ...?
```

The *`position`* can be a zero-indexed integer value, or the keyword `end` to add the tab to the end. If the *`widget`* is already managed by the notebook, it is moved to the indicated position.

You can hide a tab with the `hide` widget command, which is equivalent to setting the `-state` of that tab to `hidden`. On the other hand, the `forget` widget command removes the tab from the notebook and unmaps and unmanages the associated child widget.

The `index` widget command returns an integer index of the tab, given any supported tab identifier; it returns the number of tabs managed if you provide `end` as its argument. And the `tabs` widget command returns a list of the widgets managed by the notebook.

The notebook widget generates a `<<NotebookTabChanged>>` virtual event after a new tab is selected. See Chapter 22 for a discussion of virtual events. You can also enable keyboard traversal for a toplevel window containing a notebook widget with the following command:

```
ttk::notebook::enableTraversal notebook
```

This command automatically extends the bindings for the toplevel window containing the notebook to support the following behaviors:

- `Ctrl+Tab` selects the tab following the currently selected one
- `Shift+Ctrl+Tab` selects the tab preceding the currently selected one
- `Alt+`*`k`*, where *`k`* is the underlined character specified by the `-underline` option for a tab, selects that tab

19.4 Progressbar

A ttk::progressbar widget provides some visual feedback to a user of the status of some long-running operation. The visual appearance of the progressbar is determined by its `-orient` option, which can be `horizontal` or `vertical`, and its `-length`, which is a screen distance specifying the length of its long axis. Its behavior is determined by its `-mode` option, which can be set to `determinate` if it will indicate incremental progress to a known completion point, or `indeterminate` if you can't predict when the operation will complete.
For a determinate progressbar, the `-maximum` option is a floating-point value indicating the completion point of operation. The default value for `-maximum` is `100.0`, which is useful to represent a percentage; you could also provide a value representing an amount of time, some number of records to process, or some other benchmark. By setting the `-value` option, you can then indicate how much progress your application has made toward the maximum. You can also accomplish this by invoking the ttk::progressbar's `step` subcommand to increment the value by a given amount (which defaults to `1.0`):

progressbar step ?*amount*?

Additionally, you can use the `-variable` option to provide the name of a variable (referenced from the global namespace) with which you want the progressbar to synchronize its `-value` option.
With an indeterminate progressbar, you don't have a predictable completion point for an operation; in this case the purpose of the progressbar is only to provide some visual indication to the user that the operation is ongoing and that the application hasn't frozen. To use an indeterminate progressbar, simply invoke its `start` subcommand at the beginning of the operation and its `stop` subcommand at the end.

19.5 Separator

The ttk::separator widget displays a simple horizontal or vertical separator, as specified by its `-orient` option. You can use it to provide a visual demarcation between logical sections in a complex interface. There are no interactive controls for the separator and no additional options of note.

19.6 Sizegrip

The ttk::sizegrip widget implements a bottom right corner resize control. It allows the user to resize the containing toplevel window by pressing and dragging the grip. Proper sizegrip behavior is supported for only the bottom right corner of the toplevel.
You must explicitly grid or pack the sizegrip in the correct position on the window. For example, if you use `pack` to position the sizegrip as part of a status bar at the bottom of a toplevel, you might use code such as

```
set statusbar [ttk::frame $top.statusbar]
pack $statusbar -side bottom -fill x
set grip [ttk::sizegrip $statusbar.grip]
pack $grip -side right -anchor se
```

On the other hand, if you use `grid`, your code might look like this:

```
set grip [ttk::sizegrip $top.grip]
grid $grip -row $lastRow -column $lastColumn -sticky se
```

Note

On Mac OS X, toplevel windows automatically include a built-in sizegrip by default. Adding `ttk::sizegrip` there is harmless, since the built-in grip just masks the widget.

19.7 Treeview

The ttk::treeview widget is a powerful tool for displaying one or more columns of information, optionally in a hierarchical arrangement with dynamic collapse and expand capabilities.

19.7.1 Manipulating Treeview Items

The fundamental building block of a treeview is an *item*, which represents one line of information in the treeview. Each item has a textual label, an optional image, and an optional list of data values that are displayed in successive columns of the treeview after the label. Each item must also have

a unique identifier within the treeview; you can either supply one yourself when creating the item or have the treeview assign an identifier automatically on creation.
Items can be nested, so an item might serve as a simple leaf node, or it can serve as a parent with one or more child items beneath it. Each treeview has a *root item* created automatically when the treeview is instantiated; its identifier is the empty string. The root item itself is not displayed by the treeview. The direct children of the root item appear at the top of the hierarchy.
You add items to a treeview widget using its `insert` subcommand:

treeview insert *parent index* ?-id *id*? ?*option value* ...?

The *`parent`* argument is the identifier of the parent item for the new item. The *`index`* is the position of the new item within the parent; it can be a zero-based integer value, or the keyword `end` to place the item after all existing children of the parent. You can explicitly provide a unique string identifier for the new item with the `-id` option, or you can omit it to have the treeview automatically assign an identifier; the identifier is the return value of the command. Each item also supports a set of options that determine its appearance. You can set the options when you insert a new item, or afterward using the treeview's `item` subcommand. You can also use the `item` subcommand to query one or all of the options for an item:

treeview item *id* ?*option*? ?*value option value* ...?

The supported item options are

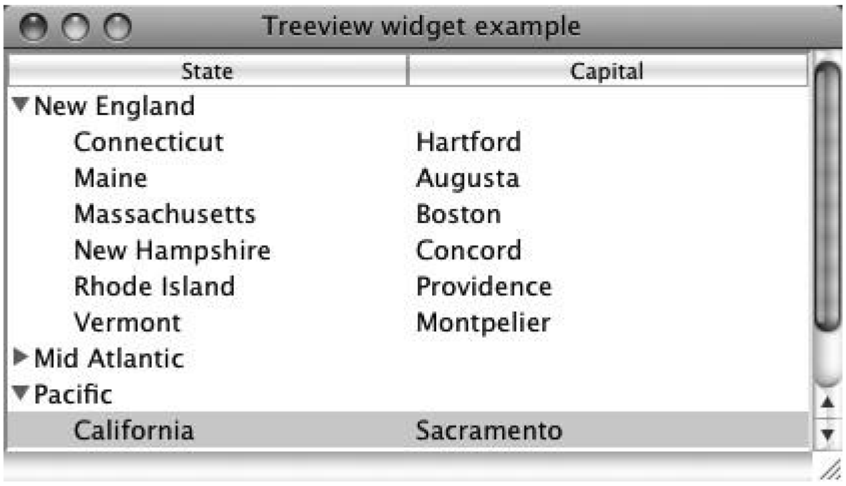
- `-text`—the text string to display for the item in the treeview
- `-image`—the name of an optional image object to display to the left of the label
- `-values`—a list of the values associated with the item
- `-open`—a Boolean value indicating whether the item's children should be displayed (true) or hidden (false)
- `-tags`—a list of tags associated with the item, as discussed below

For example, the following code creates a treeview and populates it with three toplevel items. It then creates several child items for each of the toplevel items:

```
ttk::treeview .tree -columns {capital} \
    -xscrollcommand {.hbar set} -yscrollcommand {.vbar set}
ttk::scrollbar .hbar -orient horizontal \
    -command {.tree xview}
ttk::scrollbar .vbar -orient vertical \
    -command {.tree yview}
grid .tree -row 0 -column 0 -sticky nsew
grid .vbar -row 0 -column 1 -sticky ns
grid .hbar -row 1 -column 0 -sticky ew
grid columnconfigure . 0 -weight 1
grid rowconfigure    . 0 -weight 1

.tree insert {} end -id newengland -text "New England"
.tree insert {} end -id midatlantic -text "Mid Atlantic"
.tree insert {} end -id pacific -text "Pacific"

set states {
    newengland      me  Maine           Augusta
    newengland      nh  "New Hampshire" Concord
    newengland      vt  Vermont         Montpelier
    newengland      ma  Massachusetts   Boston
    newengland      ri  "Rhode Island"  Providence
    newengland      ct  Connecticut     Hartford
    midatlantic     ny  "New York"      Albany
    midatlantic     nj  "New Jersey"    Trenton
    midatlantic     pa  Pennsylvania    Harrisburg
    pacific         wa  Washington      Olympia
    pacific         or  Oregon          Salem
    pacific         ca  California      Sacramento
}

foreach {region id state capital} $states {
    .tree insert $region end -id $id -text $state \
        -values [list $capital]
}

.tree column #0 -minwidth 150
.tree column capital -minwidth 150
.tree heading #0 -text "State"
.tree heading capital -text "Capital"
```

The resulting treeview is shown in Figure 19.4.

Figure 19.4 Example of a treeview widget



You can programmatically expand or collapse an item by setting its `-open` property to true or false, respectively. Default treeview widget class bindings automatically implement collapse/expand behavior for user interaction; clicking the left mouse button on an item toggles the item's `-open` option. The widget also generates `<<TreeviewOpen>>` and `<<TreeviewClose>>` virtual events in response to these actions, and you can implement your own custom item open/close behaviors. See Chapter 22 for more information on virtual events and executing scripts in response to events. You can also use the treeview's `see` subcommand to ensure that a specific item is visible; it sets all of the item's ancestors to `-open true` and scrolls the widget if necessary so that the item is displayed on the screen:

> *treeview* see *item*

You can use the `move` subcommand to move an existing item to a new location in the item hierarchy:

> *treeview* move *item parent index*

The *`item`* argument is the identifier of the existing item to move, and the *`parent`* and *`index`* arguments are interpreted as with `insert`. You cannot move an item under one of its descendant items.

The `delete` subcommand takes a list of one or more item identifiers and deletes each of them and all their descendant items. In contrast, the `detach` subcommand takes a list of item identifiers and "unlinks" them from the tree. The items and their descendants are no longer displayed by the tree, but they still exist. You can reattach any of them to the tree in any location by using the `move` subcommand.

You can determine the integer index of an item within its parent using the `index` subcommand. The `next` and `prev` subcommands return the identifier of an item's next or previous sibling; these return an empty string if the item is the last or first item within a parent, respectively. The `parent` subcommand returns the identifier of an item's parent, or an empty string if the item is at the top of the hierarchy. The `children` subcommand returns a list of identifiers of all of the children of an item in the order in which they appear; you can also provide a list of item identifiers, in which case the existing children are detached and the items listed become the new children.

19.7.2 Managing Treeview Columns and Headings

Your treeview can display as many columns of data as you like. The first column consists of the tree items themselves; that column displays the image associated with the item followed by its text label, as specified by the item's `-image` and `-text` properties, respectively. Both of these properties are optional. Additional columns display elements from the item's optional `-values` property. Columns also have configurable headings.
The treeview widget's `-columns` property determines the number of columns of data contained within the treeview. Its value is a list of logical column names. The default value for `-columns` is an empty list, which indicates that the treeview displays a single column containing the tree items and that the `-values` property of the items is ignored. If a list of logical column names is provided for `-columns`, the list elements of each item's `-values` property are associated with the corresponding logical column names in order. If the `-values` list contains fewer elements than logical columns, an empty string value is assumed for the remaining columns; if the list contains more elements than logical columns, the excess elements are ignored. The example in Section 19.7.1 defined a single logical column named "Capital," and each item had a one-element list for its `-values` property, providing the value for that column. The following code defines a treeview widget with three logical columns named `country`, `capital`, and `currency` and defines several items, each with three corresponding values for its `-values` property:

```
ttk::treeview .tree -columns {country capital currency}
.tree insert {} end -values {Canada Ottawa CAD}
.tree insert {} end -values {France Paris EUR}
.tree insert {} end -values {Germany Berlin EUR}
.tree insert {} end -values {Japan Tokyo JPY}
```

The order in which the logical columns are displayed does not have to be the same as the order in which they appear in `-columns`. You can also

configure the treeview to show only a subset of the logical columns. This behavior is controlled by the treeview's `-displaycolumns` property. You can set it to a list of the logical column names (or their zero-based indices) in the order in which they should be displayed. The default value for `-displaycolumns` is `#all`, which indicates that all logical columns should be displayed in the order in which they were defined.
The tree items always appear in the first display column. You cannot display them in a different column, but you can choose not to display them. The treeview's `-show` property contains a list specifying which elements of the treeview to display. The default value is `{tree headings}`, to display both the tree items and the column headings.
The `column` command queries or modifies display column configuration:

treeview column *column* ?*option*? ?*value option value* ...?

You can specify a *`column`* by the logical column name if it is displayed; you can also use the zero-based logical column index identifying the column as listed in the `-columns` property. Alternatively, you can indicate a display column with a string in the form `#`*`n`*, where *`n`* is the zero-based column number counting from the left.

Note

Column `#0` always refers to the tree column, even if it is not displayed.

The supported column options are
- `-id`—a read-only option containing the logical column name
- `-anchor`—specifies how the text in this column is aligned with respect to the cell; one of `n`, `ne`, `e`, `se`, `s`, `sw`, `w` (default), `nw`, or `center`
- `-minwidth`—the minimum width of the column in pixels; defaults to `20`
- `-stretch`—a Boolean specifying whether or not the column's width should be adjusted when the widget is resized; defaults to `1` (true)
- `-width`—the width of the column in pixels; defaults to `200`

You query or modify column headings with the `heading` command:

treeview heading *column* ?*option*? ?*value option value* ...?

The supported heading options are
- `-text`—the text to display in the column heading
- `-image`—an image to display to the right of the column heading

- `-anchor`—specifies how the heading text should be aligned; one of `n`, `ne`, `e`, `se`, `s`, `sw`, `w`, `nw`, or `center` (default)
- `-command`—a script to evaluate when the heading label is pressed

19.7.3 Treeview Item Selection Management

The default treeview widget class bindings automatically handle selecting an item when the user clicks on the item. However, the way the bindings behave depends on the value of the treeview's `-selectmode` option:

- `extended (default)`

Clicking an item selects it and deselects any other items currently selected. The user may select multiple items, using Control-clicks to toggle an item's selection and Shift-clicks to extend the selection.

- `browse`

Clicking an item selects it and deselects any other items currently selected. At most one item may be selected by the user.

- `none`

Clicking items does not affect which items are selected. Thus, this disables direct user selection of items.

The `-selectmode` option affects only the user's ability to manage treeview item selection. Regardless of the `-selectmode` value, your application can use the treeview's `selection` subcommand to control item selection:

treeview selection ?*selectOp itemList*?

Without any additional arguments, the `select` subcommand returns a list of selected items. The `selection set` subcommand causes *`itemList`* to be the new selection; providing an empty *`itemList`* removes all items from the selection. The `selection add` subcommand adds the items in *`itemList`* to the selection; the `selection remove` subcommand removes the items in *`itemList`* from the selection; and the `selection toggle` subcommand toggles the selection state of each item in *`itemList`*.

The treeview widget also generates a `<<TreeviewSelect>>` virtual event whenever an item is selected or deselected. See Chapter 22 for more information on virtual events and executing scripts in response to events.

19.7.4 Treeview Item Tags

The treeview widget allows you to define symbolic *tags* that you can apply to any number of items. Tags allow you to apply formatting to individual items in the treeview. You define, query, or modify tag properties with the `tag configure` subcommand:

treeview tag configure *tag* ?*option*? ?*value option value* ...?

The supported tag options are

- `-background`—the background color to use for the cells displaying the items
- `-foreground`—the text foreground color for the items
- `-font`—the text font for the items
- `-image`—an image to display for the items, but only if an item does not have its own `-image` property set

All tag options have a default value of the empty string, which indicates that the tag does not affect that aspect of an item. As an example, the following code defines two tags named `highlight` and `important`:

```
.tree tag configure highlight \
    -background blue -foreground white
.tree tag configure important -foreground red
```

To apply tags to an item, you provide a list of tag names as the value of the item's `-tags` property; for example:

```
.tree item vt -tags {highlight important}
```

The tags are listed in order from lowest priority to highest priority. If multiple tags have the same tag option set, the value for the tag with the highest priority is used. For example, because `important` is listed after `highlight` in the preceding line of code, its foreground color of `red` is used for the `vt` item. But because `important` does not set a value for its `-background` property, the `blue` value set by the `highlight` tag is used in this case.

The treeview widget also allows you to associate bindings to tags with the `tag bind` subcommand. You can use bindings to make items "active" so that they respond to mouse, keyboard, or virtual events. (See Chapter 22 for more information on events and executing scripts in response to events.) Among other things, this enables you to implement hypertext effects. For example, the following binding simply prints a message to the console whenever the user clicks on an item that has the tag `important`:

```
.tree tag bind important <ButtonPress-1> {
    puts "This is an important item"
}
```

Keyboard events trigger a tag binding only when the item has *focus*. At most one item at a time in a treeview has focus. The default treeview widget class bindings automatically assign focus to the last item selected by the user. You can use the `focus` subcommand to query or set the focus item:

> *treeview* focus ?*item*?

Called without an *`item`*, the command returns the identifier of the current focus item or an empty string if no item has focus. Called with an item identifier, the command assigns focus to the specified *`item`*.
It is possible for multiple bindings to match a particular event, for example, if you've created bindings for multiple tags and then applied some combination of those tags to an item. When this occurs, all of the matching bindings are invoked, in order from lowest-priority to highest-priority tag. If there are multiple matching bindings for a single tag, only the most specific binding is invoked. A `continue` command in a binding script terminates that script, and a `break` command terminates that script and skips any remaining scripts for the event, just as for the `bind` command.

Note

If bindings have been created for a treeview widget using the `bind` command, they are invoked in addition to bindings created for the tags using the `tag bind` subcommand. The bindings for tags are invoked before any of the bindings for the widget as a whole.

19.8 Themed Widget States

The *state* of a widget controls its appearance and behavior. The state of a classic widget is determined by the value of its `-state` option, as described in Section 18.15.1. Most classic widgets support only the values `normal` and `disabled`, but some widgets, such as the entry and spinbox, support additional states such as `readonly`. A classic widget can be in only one state at a time.
Themed widgets also use states, but these differ from classic widgets in two important ways. First, states consist of several independent flags, each of which may simultaneously be on or off; the complete state for the widget is

therefore equivalent to the particular combination of flags that are on or off. Second, all themed widgets support states, and all widgets use exactly the same set of state flags, though as we'll see, some flags may be ignored by particular widgets. All themed widgets currently support the following states:

- `active`

The mouse cursor is over the widget, and pressing a mouse button will cause some action to occur.

- `disabled`

The widget is disabled under program control.

- `focus`

The widget has keyboard focus.

- `pressed`

The widget is being pressed.

- `selected`

The widget is "on," "true," or "current" for things like checkbuttons and radiobuttons.

- `background`

Windows and Mac OS X have a notion of an "active" or foreground window. The `background` state is set for widgets in a background window and cleared for those in the foreground window.

- `readonly`

The widget should not allow user modification.

- `alternate`

This is a widget-specific alternate display format. For example, it is used for checkbuttons and radiobuttons in the "tristate" or "mixed" state, and for buttons that have their `-default` option set to `active`.

- `invalid`

The widget's value is invalid (for example, if an entry widget value fails validation).

For each themed widget in your application, some of these states might be "on" or "set," while the others are "off" or "unset." Default bindings for each of the themed widget classes automatically manage the states. For example, the `TButton` class (themed button) has default bindings that check whether the `disabled` state of a themed button is "off"; if so, they set the `active` state to "on" when the mouse enters the button and back to "off" when the mouse leaves the button. You can programmatically change and query the state of a themed widget using its `state` widget command:

widget state ?*stateSpec*?

The `stateSpec` is a list of states to set or unset. If an element consists of a state name, the state is set; if an element consists of a state name preceded by a `!`, the state is unset. The return value indicates which flags were changed. If you execute the `state` widget command with no `stateSpec`, it returns a list of the states currently set; for example:

```
   .e state {focus readonly}
⇒ !focus !readonly
   .e state
⇒ focus readonly
   .e state {!focus !readonly disabled}
⇒ !disabled focus readonly
   .e state
⇒ disabled
```

You can test the state of a themed widget using its `instate` widget command:

widget instate *stateSpec* ?*script*?

Without a `script` argument, the command returns `1` if the widget state matches the `stateSpec`, or `0` otherwise; for example:

```
   .e instate disabled
⇒ 1
   .e instate {disabled !readonly}
⇒ 1
   .e instate alternate
⇒ 0
```

If you provide a script argument, the script is executed if the widget state matches the state specification. For example, the following invokes a themed button, `.b`, only if it is currently pressed and enabled:

```
.b instate {pressed !disabled} { .b invoke }
```

Note

The `instate` widget command allows you to create widget or even class bindings implementing state-based behaviors quite easily. As you will see in Section 19.9.3, the `ttk::style map` command also allows you to change the appearance of widgets in particular states. In combination, this allows significant customization of the out-of-the box widget set with no need to modify the underlying code implementing the widgets.

19.9 Themed Widget Styles

Each themed widget is assigned a *style*, which specifies the appearance of the widget under different circumstances. Remember that each widget has a default class associated with it (e.g., `TLabel` for all label widgets). The default style associated with a themed widget has the same name as the widget class (e.g., a style named `TLabel`). Thus, all themed widgets of the same class have the same appearance, unless your application explicitly overrides the style for a widget.
You can use the `winfo class` command to determine the class of a particular widget; for example:

```
    ttk::label .l
    winfo class .l
⇒  TLabel
```

You can override the default style of a themed widget by assigning the style name as the value of the widget's `-style` property. (The default value is an empty string, which indicates that the widget uses the default style as defined by the class name.) For example, if you have defined a new style named `AlertLabel.TLabel`, you could assign it to a widget as follows:

```
.l configure -style AlertLabel.TLabel
```

You will see how to define new styles in the following sections.

19.9.1 Using Themes

A *theme* is a named collection of styles, designed to provide a consistent look and feel to all the themed widgets in an application. The `ttk::style theme names` command lists the names of themes defined on your current system:

```
    ttk::style theme names
⇒  aqua clam alt default classic
```

Tk automatically picks an appropriate theme for your windowing system when you launch the application. The `ttk::currentTheme` variable contains the name of the current theme:

```
    puts $ttk::currentTheme
⇒ aqua
```

You can switch to a different theme by executing `ttk::style theme use`:

```
ttk::style theme use clam
```

This command updates all styles with the definitions from the new theme and refreshes all themed widgets to reflect the new theme.

19.9.2 The Elements of Style

A themed widget is implemented internally as a collection of *elements*. One function of a style is to control the placement of elements.

Note

In general, application developers rarely modify widget layouts. However, it can be useful to determine the elements that compose a particular themed widget when you want to customize its configuration, as discussed in the next section.

You can use the `ttk::style layout` command to view or change the layout of elements for a given style. Here is an example under the Aqua theme:

```
  ttk::style layout TCheckbutton
⇒ Checkbutton.button -sticky nswe -children {Checkbutton.padding
  -sticky nswe -children {Checkbutton.label -side left -sticky {}}}
```

A layout consists of a list of elements, each followed by one or more options specifying how to arrange the elements. In the return value above, `Checkbutton.button`, `Checkbutton.padding`, and `Checkbutton.label` are the elements composing a themed checkbutton widget under the Aqua theme. Reformatting the output can make the structure clearer:

```
Checkbutton.button -sticky nswe -children {
    Checkbutton.padding -sticky nswe -children {
        Checkbutton.label -side left -sticky {}
    }
}
```

The elements are arranged in the order listed within the space allocated to the widget, known as the *cavity*. The optional `-side` property determines the location of the *parcel* of space allocated to the element, and it can be set to one of `left`, `right`, `top`, or `bottom`. If you omit the `-side` property, the entire cavity is allocated to the element. The `-sticky` option specifies where the element resides in its parcel by indicating the sides of the parcel to which the element "sticks." For example, the `-sticky nswe` option in the preceding example indicates that the element should fill all of the parcel. The `-children` property specifies a list of one or more elements arranged inside the parent element, following the same rules discussed before.
Note that the layout for a particular style might vary from one theme to another. This is because some windowing systems use elements on widgets (such as focus indicators) that aren't used by other windowing systems. For example, under the Aqua theme, the `TButton` layout is defined as follows:

```
Button.button -sticky nswe -children {
    Button.padding -sticky nswe -children {
        Button.label -sticky nswe
    }
}
```

In contrast, the `TButton` layout has the following definition under the XP native theme:

```
Button.button -sticky nswe -children {
    Button.focus -sticky nswe -children {
        Button.padding -sticky nswe -children {
            Button.label -sticky nswe
        }
    }
}
```

19.9.3 Creating and Configuring Styles

As an application developer, you might want to change the appearance of various widgets in your application, for example, setting the foreground

color used to render text in a set of buttons. Rather than requiring you to set the configuration options on each individual widget, themed widgets allow you to set the configuration options for the style, which then takes effect automatically for each widget that uses that style.
Before you can configure the options associated with a style, you need to know what options are available on that style. Each element that is a component of a themed widget has a set of options that it uses. You can retrieve a list of the options associated with a given element using the `ttk::style element options` command, like so:

```
  ttk::style element options TLabel.label
⇒ -compound -space -text -font -foreground -underline -width -anchor
  -justify -wraplength -embossed -image -stipple -background
```

The actual values for element options are supplied by the style containing the element, or in some cases (for example, `-text`) by the actual instance of a themed widget. When configuring a style, you can provide values for any of the options associated with any of the style's elements. A single option, such as `-background`, might be associated with several elements within a style. In that case, the value provided by the style would be used by all of the associated elements. This feature ensures visual consistency in the widget.
The `ttk::style configure` command queries or modifies the option values of a style:

ttk::style configure *style* ?*option*? ?*value option value* ...?

For example, you could configure the `TLabel` style so that every themed button in your application displays its text in green:

```
ttk::style configure TLabel -foreground green
```

More typically, you might want a customization to apply to only some of the widgets of a particular class in your application. For example, you might want most of your labels to use the standard colors defined by the theme, but some labels are special alerts that should use red text. In that case, you can define a new style derived from the `TLabel` style, as in the following example:

```
ttk::style configure AlertLabel.TLabel -foreground red
```

The style name `AlertLabel.TLabel` indicates that it is derived from `TLabel`. If an element queries the `-foreground` option, it gets the value `red`; if it queries any other option, it gets any value configured in the `TLabel` base style. You can create as deep a hierarchy of derived styles as you like, using a `.` to separate

each level. When an element queries an option, it obtains the value from the first style where it is defined. There is also a base style named `.` that is used to provide default option values for all styles. The primary purpose of the `.` style is to serve as a fundamental building block for defining the look and feel of a theme.
The option values you set with the `ttk::style configure` command are the default values that apply. However, you can also use the `ttk::style map` command to define override values to use depending on the state of the widget:

ttk::style map *style* ?*option*? ?{*stateSpec value* ...} ...?

The *`style`* and *`option`* arguments are interpreted as with `ttk::style configure`. The next argument is a list of state specifications (expressed in the same format as described in Section 19.8) and corresponding option values. The actual state of a particular widget is compared against the state specifications in the order in which they are listed. The first specification that matches the widget's state determines the value to use for the option. State maps are inherited as well, so if there is no appropriate mapping for an option in the current style, the style from which it was derived is then checked, and so on. If none of the state specifications matches, the option gets the default value as defined by the `ttk::style configure` definitions described earlier.
The `ttk::style lookup` command allows you to retrieve the value of a particular option for a style:

ttk::style lookup *style option* ?*stateSpec* ?*default*??

With no *`stateSpec`* or *`default`* arguments, the command returns the default value, as defined by `ttk::style configure`. For example, to retrieve the default button font:

```
   ttk::style lookup TButton -font
⇒ TkDefaultFont
```

Given a *`stateSpec`*, the command returns a value using the standard lookup rules for element options (that is, checking for a matching `ttk::style map` state specification first and falling back to the defaults defined by `ttk::style configure` if there is no matching state specification). If the *`default`* argument is present, it is used as a fallback value in case no specification is found for the option.

19.10 Other Standard Themed Widget Options

Like classic widgets, themed widgets support options that control their appearance and behavior. The configuration options for a themed widget may be specified when the widget is created and modified later with the `configure` widget command; you can query the value of a widget's configuration option using the `cget` widget command.

The standard set of options supported by themed widgets is much smaller than the set for classic widgets. Unlike classic widgets, most themed widgets don't support options such as `-background`, `-relief`, or `-font` for controlling color, borders, or fonts. Instead, these visual characteristics are controlled by the widget's *style*, which is usually determined by the overall *theme* selected for the application. Styles are discussed more in Section 19.9.

Themed widgets support a `-cursor` option, which controls the appearance of the mouse cursor when the mouse is over the widget, and it works just as with classic widgets, as discussed in Section 18.15.5. Scrollable themed widgets also support `-xscrollcommand` and `-yscrollcommand` options and follow the same scrolling protocols described in Section 18.9.

Labels, buttons, and other button-like themed widgets support `-text` and `-textvariable` options that behave like their classic counterparts. These widgets also support a `-image` option for specifying the name of an image object to display. However, rather than being limited to a single image name, themed widgets allow you to specify a list of values for the `-image` option. The first element is the name of the default image to display. Subsequent elements are two-element nested lists, where the first element is a state specification, as defined in Section 19.8, and the second element is the name of an image to display in that state. The first state specification that matches the widget's current state determines the image to display. All images should have the same size.

The `-compound` option determines how to position the image relative to the text, if both have been specified for a widget. The default of `none` displays the image if present, otherwise the text. A value of `text` or `image` displays only one or the other type of content. A value of `top`, `bottom`, `left`, or `right` places the image in the indicated position relative to the text. A value of `center` displays the text centered on top of the image.

20. Fonts, Bitmaps, and Images

The primary function of any graphical interface is to present text and pictures, and the resources needed to accomplish this are fonts and images. Available fonts vary greatly from one platform to another, and even from one Windows PC to another. Having a way to manage font usage in an application simplifies supporting multiple platforms and multiple environments. Tk provides the `font` command to create named fonts and to manage font use in applications. Likewise, icons have become as important as any glyph in any alphabet in today's applications. These images, some of which are common and some of which are unique, are necessary in any application with a graphical user interface. The Tk `image` command is used to load, manipulate, and share image data within applications. In this chapter we present these two commands.

20.1 Commands Presented in This Chapter

This chapter discusses the following commands for manipulating fonts and images:

- `font actual` *`font`* `?-displayof` *`window`*`? ?`*`option`*`? ?--? ?`*`char`*`?`

Returns the actual attribute values for a font. The actual values can vary from what was specified because of platform limitations. The options are the same as those supported for `font configure`.

- `font configure` *`fontname`* `?`*`option`*`? ?`*`value option value`* `...?`

Queries or configures the attributes for a named font. If a single *`option`* is specified with no *`value`*, it returns the current value of that attribute. If one or more *`option value`* pairs are specified, the command modifies the given named font to have the given values. See the text and the reference documentation for a description of the supported attributes and their values.

- `font create ?`*`fontname`*`? ?`*`option value`* `...?`

Creates a new named *`fontname`* with the attributes specified by the *`option value`* pairs and returns its name. If *`fontname`* is omitted, Tk generates a new name. The options are the same as those supported for `font configure`.

- `font delete` *`fontname`* `?`*`fontname`* `...?`

Deletes the specified named fonts.

- `font families ?-displayof` *`window`*`?`

Returns a list of the case-insensitive names of all font families that exist on *window*'s display.

- `font measure` *font* `?-displayof` *window*`?` *text*

Measures the amount of space the string *text* would use in the given font when displayed in *window*. The return value is the total width in pixels of *text*, not including the extra pixels used by highly exaggerated characters such as cursive *f*. Format control characters like newline and tab are ignored.

- `font metrics` *font* `?-displayof` *window*`?` `?`*option*`?`

Returns information about the metrics for *font* when it is used on *window*'s display. See the reference documentation for more information.

- `font names`

Returns a list of all the named fonts currently defined.

- `image create` *type* `?`*name*`?` `?`*option value* `...?`

Creates a new image of *type* `bitmap` or `photo`. A name is created automatically if *name* is omitted. Returns the name of the image created. Also creates a command of the same name for further image access. See the text and the reference documentation for information on the supported options.

- `image delete ?`*name* `...?`

Deletes each of the named images. Each instance where the image is displayed retains its size, but the area goes blank. The actual image is not deleted until all the instances are released.

- `image height` *name*

Returns the height of the image in pixels.

- `image inuse` *name*

Returns a Boolean value indicating whether the image is in use by any widgets.

- `image names`

Returns a list containing the names of all existing images.

- `image type` *name*

Returns the type of the image *name*, for example, `bitmap` or `photo`.

- `image types`

Returns a list whose elements are all the valid values for the type parameter to the `image create` command.

- `image width` *name*

Returns the width of the image in pixels.

20.2 The `font` Command

While Mac OS X and Windows specify fonts using the same nomenclature, X-based windowing systems use a distinctly different naming scheme. With X, the font name specifies a number of parameters, such as

```
-adobe-times-bold-r-normal--18-180-75-75-p-99-iso8859-1
```

This font name specifies a Times Bold font from Adobe, at 18 point, designed for a 75 DPI screen and supporting the Western European ISO-8859-1 character set. Mac OS X and Windows specify fonts using much shorter names, like the following:

```
Times Bold 18
```

Tk abstracts out this difference by using *named fonts*. Tk widgets accept fonts specified in any one of five different forms; however, the best way to use Tk is with named fonts. In this way, platform variations are easily isolated.

Note

If the master font that you set for a widget doesn't contain a glyph for a particular Unicode character that you want to display, Tk attempts to locate a font that does. Where possible, Tk attempts to locate a font that matches as many characteristics of the widget's master font as possible (for example, weight, slant, etc.). Once Tk finds a suitable font, it displays the character in that font. In other words, the widget uses the master font for all characters it is capable of displaying and alternative fonts only as needed. In some cases Tk is unable to identify a suitable font, in which case the widget cannot display the characters. Instead, the widget displays a system-dependent fallback character such as `?`.

Tk comes with a predefined set of named fonts on all platforms that match appropriate system defaults. You should not change these fonts, as Tk itself may modify them in response to system changes. The font names and their default/recommended uses are as follows:

- `TkDefaultFont`—the default for all GUI items not otherwise specified
- `TkTextFont`—font used for user text in entry widgets, listboxes, etc.
- `TkFixedFont`—the standard fixed-width font
- `TkMenuFont`—font used for menu items
- `TkHeadingFont`—font used for column headings in lists and tables
- `TkCaptionFont`—font used for window and dialog caption bars

- `TkSmallCaptionFont`—font used for captions on contained windows or tool dialogs
- `TkIconFont`—font used for icon captions
- `TkTooltipFont`—font used for tooltip windows (transient information windows)

20.2.1 Manipulating and Using Named Fonts

To use a named font, you first create a new font, giving it the name you want to use. When you create the font, you specify the font family, size, and so on, using the `font create` command:

```
font create ?fontname? ?option value ...?
```

You can either provide a specific *`fontname`* for the named font, or you can have `font create` automatically assign a name consisting of `font` followed by a unique integer. The return value is the name of the font created. The *`option value`* pair arguments that follow specify the characteristics of the font. You can use any of the following options to configure the font:

- `-family` *`name`*

The case-insensitive font family *`name`*. Tk guarantees to support the font families named Courier, Times, and Helvetica. The most closely matching native font family is substituted automatically when one of these font families is used. The name may also be the name of a native, platform-specific font family. If the family is unspecified or unrecognized, a platform-specific default font is chosen.

- `-size` *`size`*

The desired size of the font. A positive size argument is interpreted as a size in points. A negative size number is interpreted as a size in pixels. If a font cannot be displayed at the specified size, a nearby size is chosen. If *`size`* is unspecified or zero, a platform-dependent default size is chosen.

- `-weight` *`weight`*

The nominal thickness of the characters in the font. Valid values are `normal` to specify a normal-weight font and `bold` to specify a bold font.

- `-slant` *`slant`*

The amount the characters in the font are slanted away from the vertical. Valid values for *`slant`* are `roman` and `italic`.

- `-underline` *`boolean`*

The value is a Boolean flag that specifies whether characters in this font should be underlined.

- `-overstrike` *`boolean`*

The value is a Boolean flag that specifies whether a horizontal line should be drawn through the middle of characters in this font.
For example, the following creates a font called `LobsterBold` based on the Helvetica font:

```
font create LobsterBold –family Helvetica \
        -size 12 -slant roman -weight bold
```

After you have created a named font, you can use it anywhere that Tk accepts a font, such as the `-font` attribute of particular widgets or the `-font` attribute of tags in a text widget:

```
.mylabel configure -font LobsterBold
```

After you have defined a named font, you can later modify its definition using the `font configure` command:

font configure *fontname ?option? ?value option value ...?*

You can change any of the attributes of a named font by providing values for the specified options. The options supported are the same as for `font create`. If you provide the name of only one option with no new value, the `font configure` command returns that attribute's current value. If you provide only the font name as an argument, the command returns a dictionary of all font attributes and their values.
When you use `font configure` to change the definition of a named font, all widgets in your application that use the named font immediately update to reflect the change. Thus, the following command would cause all widgets that use the `LobsterBold` font to reduce the text they display to 10 points:

```
font configure LobsterBold -size 10
```

The `font names` command returns a list of all named fonts. The `font actual` command allows you to determine the actual attributes of the named font. These might differ from the requested values if, for example, the requested font family or other font attributes are not available on the local system. The following example shows how the Arial font was substituted for Helvetica in the definition of `LobsterBold` on a Windows system:

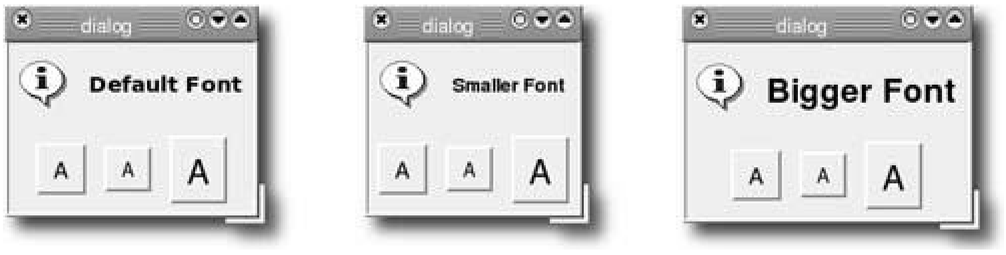
```
  font actual LobsterBold
⇒ -family Arial -size 10 -weight bold -slant roman -underline 0
  -overstrike 0
```

The `font delete` command deletes one or more named fonts. If there are widgets using a named font that has been deleted, the named font isn't actually deleted until all the instances are released. Those widgets continue to display using the last known values for the named font. If a new named font is created with the same name while there are widgets still using the former named font, the widgets redisplay using the new attributes of the font. The following script shows how to create named fonts in Tk as well as change the size of the font:

```
#
# Demonstrate named fonts
#

set top [toplevel .dialog]
set icon [label $top.icon -bitmap info]
set l [label $top.msg -textvariable message]
set message "Named Font Demonstration"
set defaultFont [$l cget -font]
set f [frame $top.f]

set bDefault [button $f.def -text A -command \
    "configDefault" -font {helvetica}]
set bSmaller [button $f.sm -text A -command \
    "configSize -2" -font {helvetica 10}]
set bBigger  [button $f.bg -text A -command \
    "configSize +2" -font {helvetica 18}]

pack $bDefault $bSmaller $bBigger -side left -pady 5 -padx 5
grid $icon     -row 0 -column 0 -padx 5 -pady 10
grid $l        -row 0 -column 1 -sticky ew -padx 5 -pady 10
grid $f     -  -row 1

set defaultSize   [font actual $defaultFont -size]
set defaultFamily [font actual $defaultFont -family]
set defaultWeight [font actual $defaultFont -weight]
set size   18
set family Helvetica
set weight bold
font create LobsterBold -size $size -family $family \
     -weight bold
$1 configure -font LobsterBold

proc configDefault {} {
    font configure LobsterBold \
        -family $::defaultFamily \
        -size $::defaultSize \
        -weight $::defaultWeight
    set ::message "Default Font"
}

proc configSize {delta} {
    font configure LobsterBold \
      -family Helvetica \
      -size [expr {[font actual LobsterBold -size] + $delta}]
    if {$delta < 0} {
        set ::message "Smaller Font"
    } else {
        set ::message "Bigger Font"
    }
    puts "font size is [font actual LobsterBold -size]"
}
```

Figure 20.1 shows the result of running this script, with three different screen shots reflecting the results of pressing the buttons to change the font configuration.

Figure 20.1 Examples of changing named font configuration



20.2.2 Other Font Utilities

In addition to manipulating named fonts, the `font` command provides additional subcommands for querying information about fonts.
The `font families` command returns a list of all font families that are available. For systems with multiple displays, different sets of fonts might

be available on different displays, so the command supports the `-displayof` option, allowing you to provide a widget name to retrieve the list of font families available on the display where the widget appears. The command defaults to using the main display if you don't provide the `-displayof` option. The `font families` command is useful for creating font selector menus or dialogs to allow the user to choose a font for use in your application.
The `font measure` command returns the width in pixels that a string would use if displayed in a given font:

```
    font measure times "This is a text"
⇒67
```

20.2.3 Font Descriptions

As mentioned earlier, fonts have five possible formats in Tk. Any of these may be used wherever a font may be specified, either in a window property, in a text or canvas widget tag property, or in the `font` command. The five formats are as follows:

- *`fontname`*

The name of a named font, created using the `font create` command. Named fonts always work without error. If the named font cannot be displayed with exactly the specified attributes, some other close font is substituted automatically.

- *`systemfont`*

The platform-specific name of a font, interpreted by the windowing system. This also includes an XLFD under X (see below). Tk provides a set of predefined named fonts that are mapped to system fonts. See the reference documentation for details.

- *`family`* `?`*`size`*`? ?`*`style`*`? ?`*`style`*` ...?`

A properly formed list where the first element is the font family and the optional second element is the size. The *`size`* attribute follows the same rules described for the `-size` attribute for `font configure`. Any additional arguments following the size are font styles. Possible values for *`style`* are `normal`, `bold`, `roman`, `italic`, `underline`, and `overstrike`.

- X-font names (XLFD)

A Unix-centric font name of the form

-foundry-family-weight-slant-setwidth-addstyle-pixel-point
-resx-resy-spacing-width-charset-encoding

The `*` character may be used to designate default values for a field. There must be exactly one `*` for each default field, except that a `*` at the end of the XLFD defaults any remaining fields; the shortest valid XLFD is simply `*`, signifying all fields as defaults.

- *`option value ?option value ...?`*

A properly formed list of *`option value`* pairs that specify the attributes of the font, in the same format as is used with `font configure`.

When a font is specified, Tk parses it and attempts to identify the font using the five preceding rules, in order. For the first two cases, the name must match exactly. For the remaining three cases, the closest available font is used and, failing that, a system-dependent default is chosen. If a font description does not match any of these patterns, an error is generated.

20.3 The `image` Command

The Tk `image` command is used to create and manage images. It works by creating an *image object*, which is then passed to widgets via the `-image` or `-bitmap` options or manipulated through the object's subcommands. When an image is created, a new command is created whose name is the same as the name of the image. This command is then used to manage the pixel data in the image.

Note

By default, the `image` command is created in the global namespace. The newly created `image` command also overwrites any existing command of that same name. Be careful when selecting image names not to overwrite existing commands, or else let `image create` assign image names for you. One way to manage image names is to use fully qualified namespace names for the images, as discussed in Section 20.3.3.

There are two built-in types of images: `bitmap`, for two-color images, and `photo`, for multicolor images. New types can be defined using the `Tk_CreateImageType` C API as an extension. The Img extension, for example, defines a type `pixmap` for X11 pixmap images. The `photo` image type can also

be extended via the `Tk_CreatePhotoImageFormat` C API, to provide support of different data formats. The Img extension does this as well to support a variety of formats other than those built into Tk.
A bitmap image is defined with a simple bit plane specifying on/off or foreground/background colors coupled with a mask. The mask defines the area of the image to be displayed while the rest of the area is transparent, where the underlying widget shows through. There are a few places in Tk where only bitmap images can be used, for example, the `wm iconbitmap` command. This command defines the icon used in the windowing system when the window is minimized. (Some windowing systems allow only simple bitmaps in this situation.)
The second type of image is a photo. The photo image is a full-color (32 bits per pixel) image internally and is displayed using dithering if necessary. Tk has built-in support for images created from GIF and PPM/PGM file formats, and extensions add support for other formats. The two built-in image types are discussed in more detail in the following sections.

20.3.1 Bitmap Images

You create bitmap images with the `bitmap` type argument to `image create`:

image create bitmap ?*name*? ?*option value* ...?

You can either provide a specific *`name`* for the bitmap or have `image create` automatically assign a name consisting of `image` followed by a unique integer. The return value is the name of the bitmap image created. The *`option value`* pair arguments that follow specify the characteristics of the bitmap. You can use any of the following options to configure the bitmap:

- `-background` *`color`*

Specifies a background color for the image. If this option is set to an empty string, the background pixels are transparent. This effect is achieved by using the source bitmap as the mask bitmap, ignoring any `-maskdata` or `-maskfile` options.

- `-data` *`string`*

Specifies the contents of the source bitmap as a string in the X11 bitmap format. If both the `-data` and `-file` options are specified, the `-data` option takes precedence.

- `-file` *`fileName`*

Gives the name of a file whose contents define the source bitmap in the X11 bitmap format.

- `-foreground` *`color`*

Specifies a foreground color for the image.

- `-maskdata` *`string`*

Specifies the contents of the mask as a string in the X11 bitmap format. If both the `-maskdata` and `-maskfile` options are specified, the `-maskdata` option takes precedence.

- `-maskfile` *`fileName`*

Gives the name of a file whose contents define the mask in the X11 bitmap format.
For the bitmap type, the image command returned by `image create` supports two subcommands, `cget` and `configure`. The `cget` subcommand returns the current value of the specified option. The `configure` subcommand queries or modifies the options specified or returns a list of all the options and their values if no arguments are given. A bitmap image can be used with any widget that supports a `-bitmap` option.

20.3.2 Photo Images

Photo images are full-color images. The `image create photo` command creates an image object and a Tcl command for manipulating the image. The `image create` command accepts these options for the photo type:

- `-data` *`string`*

Specifies the contents of the image as a string. The string can contain base64-encoded data or binary data. The format of the string must be one of those for which there is an image file format handler that accepts string data.

- `-format` *`formatName`*

Specifies the name of the file format for the data specified with the `-data` or `-file` option.

- `-file` *`fileName`*

Gives the name of a file containing data for the photo image. The file format must be one of those for which there is an image file format handler that can read data.

- `-gamma` *`value`*

Specifies the gamma correction to apply to the color map for this image. The value specified must be greater than zero. The default value is `1` (no correction). In general, values greater than `1` make the image lighter, and values less than `1` make it darker.

- `-height` *`number`*

Specifies the height of the image, in pixels. A value of `0` (the default) allows the image to expand or shrink vertically to fit the data stored in it.

- `-palette` *`paletteSpec`*

Specifies the color palette to use for this image. The *`paletteSpec`* string may be a single decimal number specifying the number of shades of gray to use, for a monochrome image. Or the *`paletteSpec`* may be three decimal numbers separated by slashes (`/`), specifying the number of shades of red, green, and blue to use, respectively, for a color image.

- `-width` *`number`*

Specifies the width of the image, in pixels. A value of `0` (the default) allows the image to expand or shrink horizontally to fit the data stored in it.
The image command returned supports a number of subcommands. For those commands that write data to the image, the image size can be adjusted as necessary, unless the `-width` and `-height` options have been configured with nonzero values. Several of the subcommands accept additional options. Some of the options are discussed in the examples that follow; read the reference documentation for a complete description.

- *`imageName`* `blank`

Erases the image, making it entirely transparent.

- *`imageName`* `cget` *`option`*

Returns the current value of the configuration option given by *`option`*.

- *`imageName`* `configure ?`*`option`*`? ?`*`value option value ...`*`?`

Queries or modifies the configuration options for the image.

- *`imageName`* `copy` *`sourceImage`* `?`*`option value ...`*`?`

Copies a region from *`sourceImage`* to *`imageName`*. You can optionally specify rectangular subregions of the source and/or destination image, as well as optionally cropping, subsampling, or zooming, and whether the source image replaces or overlays the destination.

- *`imageName`* `data ?`*`option value ...`*`?`

Returns image data in the form of a string. You can optionally specify a rectangular subregion of the image to return, the data format, whether transparent pixels are replaced with a color, and whether to transform the data into grayscale.

- *`imageName`* `get` *`x y`*

Returns the color of the pixel at coordinates *`x`*,*`y`* in the image as a list of three integers between 0 and 255, representing the red, green, and blue components respectively.

- *`imageName`* `put` *`data`* `?`*`option value ...`*`?`

Sets pixels in *`imageName`* to the data specified in *`data`*. You can optionally specify a rectangular subregion for the target data and the image format of the data string. The data can also be specified as a Tcl list of scan lines, with each scan line consisting of a list of pixel colors.

- *imageName* read *fileName* ?*option value* ...?

Reads image data from the file named *fileName* into the image. You can optionally specify rectangular subregions of the source and/or destination image, as well as whether *imageName* is cropped to fit the data read.

- *imageName* redither

Recalculates the dithered image in each window where the image is displayed.

- *imageName* transparency get *x y*

Returns a Boolean indicating whether the pixel at *x*,*y* is transparent.

- *imageName* transparency set *x y boolean*

Makes the pixel at *x*,*y* transparent if *boolean* is true and makes that pixel opaque otherwise.

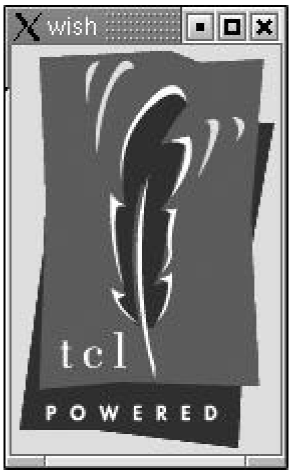
- *imageName* write *fileName* ?*option value* ...?

Writes image data from *imageName* to a file named *fileName*. You can optionally specify a rectangular subregion of the image to return, the data format, whether transparent pixels are replaced with a color, and whether to transform the data into grayscale.

The following examples demonstrate how photo images can be loaded and manipulated with the `image` command using the Tcl Powered logo, a handy sample image found in the Tk library.

The first script reads the Tcl Powered logo image and displays it in a label. The result is shown in Figure 20.2.

Figure 20.2 The Tcl Powered logo image

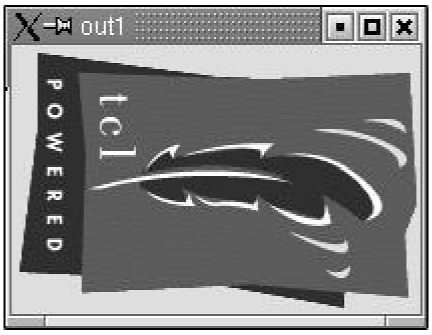


```
# Fetch and display a source image
set img [image create photo \
    -file $tk_library/images/pwrdLogo200.gif]
label .img -image $img -anchor nw
grid .img -sticky nw
```

The second script creates a new image in which the Tcl Powered logo is rotated 90 degrees. It starts by creating an empty image object. The data from the original image is then read a pixel at a time with the `get` subcommand. The return value is a three-element list of the red, green, and blue color values for the pixels, ranging from 0 to 255. This is converted into a standard `#RRGGBB` color string and written to the appropriate pixels in the new image using the `put` subcommand. To handle transparency properly, the transparency setting of the original pixel is tested and replicated on the target image if necessary. The result is shown in Figure 20.3.

```
# Rotate image 90 degrees clockwise
set img1 [image create photo]
toplevel .out1
label .out1.img -image $img1 -anchor nw \
    -height [image width $img] \
    -width [image height $img]
grid .out1.img -sticky nw

set wid [image width $img]
set hgt [image height $img]
for {set x 0} {$x < $wid} {incr x} {
  for {set y 0} {$y < $hgt} {incr y} {
    set color [$img get $x $y]
    set rgb [format {#%02x%02x%02x} {*}$color]
    $img1 put $rgb -to [expr {$hgt - $y}] $x
    if {[$img transparency get $x $y]} {
      $img1 transparency set [expr {$hgt - $y}] $x 1
    }
  }
}
```

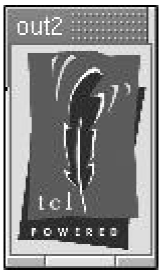
Figure 20.3 The image rotated clockwise 90 degrees



The third script creates a new image that is reduced to half the size of the original. It does so by creating a new empty image object, then it uses the `copy` subcommand to copy the image data from the original to the new image. The `-subsample` option accepts two additional arguments indicating that the copy should use only every *x*th and *y*th pixel in the x- and y-coordinates, respectively. By setting these values to `2` and `2`, only every other pixel in the `x` and `y` directions is copied. The result is shown in Figure 20.4. Although this is fast, it can result in blocky-looking images. A better-looking result usually is achieved by averaging pixel values, as is shown next.

```
# Create half size image by subsampling
set img2 [image create photo]
toplevel .out2
label .out2.img -image $img2 -anchor nw
grid .out2.img -sticky nw
$img2 copy $img -subsample 2 2
```

Figure 20.4 The image reduced to half size using subsample copying



The last script also creates a new half-size image. In this case, it does so by reading four-pixel blocks using the `get` subcommand, averaging the color values, and writing the result to the target image. If three or four of the pixels are transparent, the result is set to transparent. The result is shown in Figure 20.5. This averaging technique usually produces a better-looking result than subsampling, but it takes substantially longer to process.

```
# Create half-size image
# by averaging 4-pixel blocks
set img3 [image create photo]
toplevel .out3
label .out3.img -image $img3 -anchor nw
grid .out3.img -sticky nw

set wid [image width $img]
set hgt [image height $img]
for {set y 0} {$y < ($hgt - 1)} {incr y 2} {
  set dy [expr {$y / 2}]
  for {set x 0} {$x < ($wid - 1)} {incr x 2} {
    set r 0
    set g 0
    set b 0
    set n 0
    for {set i 0} {$i < 2} {incr i} {
      set xi [expr {$x+$i}]
      for {set j 0} {$j < 2} {incr j} {
        set yj [expr {$y+$j}]
        if {[$img transparency get $xi $yj]} {
          continue


        }
        incr n
        set cx [$img get $xi $yj]
        incr r [lindex $cx 0]
        incr g [lindex $cx 1]
        incr b [lindex $cx 2]
      }
    }
    set dx [expr {$x / 2}]
    if {$n < 2} {
      $img3 put "#000000" -to $dx $dy
      $img3 transparency set $dx $dy 1
    } else {
      set r [expr {$r / $n}]
      set g [expr {$g / $n}]
      set b [expr {$b / $n}]
      set color [format "#%02x%02x%02x" $r $g $b]
      $img3 put $color -to $dx $dy
    }
  }
}
```

Figure 20.5 The image reduced to half size using pixel averaging

Tk has a C language interface to the `image` command that can be used to implement support for other image file types. The Img package does just that, providing support for BMP, ICO, JPEG, PCX, PNG, PPM, PS, SGI, SUN, TGA, TIFF, XBM, and XPM file types. This extension also has an option to capture any Tk window as an image.

Another extension is the TkMagick/TclMagick package that supports approximately 100 types. It also provides extensive image manipulation capabilities, which can be used in Tcl without Tk. This facilitates image manipulation in server-side applications.

The examples in this section simply illustrate how the `image` commands work. Transformations like the one shown in Figure 20.5 can be performed more efficiently with an extension like `TclMagick`. The basic Tk `image` command simply provides a standardized interface to extensions like Img and TkMagick so that complex image manipulation can be controlled easily from Tcl and the results displayed using Tk.

20.3.3 Images and Namespaces

Because creating image objects also creates image commands with the same name, you must be careful not to overwrite existing command names. One solution is to let `image create` automatically name the images, in which case they'll have names like `image0`, `image1`, and so on. By default, all image commands are created in the global namespace.

Another strategy is to create a separate namespace for your image names, such as `::img`, and then create your images with fully qualified namespace names, such as `::img::logo` and `::img::large`. With this technique, any images associated with a particular module or library could also be created within the library's namespace. See Chapter 10 for more information on namespaces.

Note

For this technique to work, you should always use fully qualified namespace references for your image names, such as `::img::logo`, not partially qualified references like `img::logo`. The image command is created relative to the current namespace, so if you are not in the global namespace when you create the image and you use a partially qualified reference for the image name, the name of the image command does not match the name of the image itself.

21. Geometry Managers

Geometry managers determine the sizes and locations of widgets. Tk is similar to other toolkits in that it does not allow widgets to determine their own geometries. A widget does not even appear on the screen unless it is managed by a geometry manager. This separation of geometry management from internal widget behavior allows multiple geometry managers to exist simultaneously and permits any widget to be used with any geometry manager. If widgets controlled their own geometry, this flexibility would be lost: every existing widget would need to be modified to introduce a new style of layout.

This chapter describes the overall structure of geometry management and then presents the three geometry managers. The *gridder*, which is the most commonly used geometry manager in Tk, allows you to arrange widgets into rows and columns. The *packer* arranges widgets around the edges of an area. The *placer* provides simple fixed and "rubber sheet" placement of widgets. There are also several widget classes that can act as geometry managers: `canvas` and `text`, which are discussed in Chapter 23 and Chapter 24 respectively; `panedwindow` (and its themed counterpart, `ttk::panedwindow`), which is described in Chapter 18; and `ttk::notebook`, which is described in Chapter 19.

21.1 Commands Presented in This Chapter

All geometry managers have a common set of subcommands, as well as a set of unique features. The following are the subcommands they have in common, where *`gm`* is one of `place`, `pack`, or `grid`:

- *`gm slave ?slave ...? option value ?option value ...?`*

Same as the `configure` subcommand described below, unless you are querying geometry manager options.

- *`gm`* `configure` *`slave ?slave ...? ?option?`*

  `?value option value ...?`

Arranges for the geometry manager to manage the geometry of the widgets named by the *`slave`* arguments. The *`option`* and *`value`* arguments provide information that determines the dimensions and positions of the slaves.

- *`gm`* `forget` *`slave ?slave ...?`*

Causes the geometry manager to stop managing the widgets named by the `slave` arguments and unmap them from the screen. Has no effect if `slave` isn't currently managed by this geometry manager.

- *gm* `info` *slave*

Returns a list giving the current configuration of `slave`. The list consists of *option-value* pairs in exactly the same form as might be specified to the geometry manager's `configure` command. Returns an empty string if `slave` isn't currently managed by this geometry manager.

- *gm* `slaves` *master* ?*option value*?

Returns a list of the slaves on `master`'s list for this geometry manager.
This chapter also discusses the following commands for managing the stacking order of widgets:

- `lower` *widget* ?*belowThis*?

Changes `widget`'s position in the stacking order so that it is just below the widget given by `belowThis`. If `belowThis` is omitted, it moves `widget` to the bottom of the stacking order.

- `raise` *widget* ?*aboveThis*?

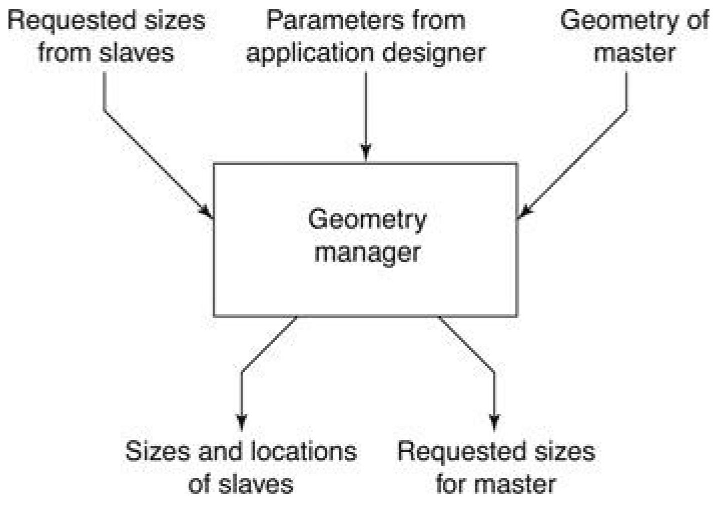
Changes `widget`'s position in the stacking order so that it is just above the widget given by `aboveThis`. If `aboveThis` is omitted, it moves `widget` to the top of the stacking order.

21.2 An Overview of Geometry Management

A geometry manager's job is to arrange one or more *slave* widgets relative to a *master* widget. For example, it might arrange three slaves in a row from left to right across the area of the master, or it might arrange two slaves so that they split the space of the master, with one slave occupying the top half and the other occupying the bottom half. Different geometry managers embody different styles of layout. The master is usually the parent of the slave, but there are times when it's convenient to use other windows as masters (you will see examples of this later).
Masters are usually frame or toplevel widgets but can be any widget. A master is used to define the geometric bounds for the slave widgets only, and the widgets' contents are otherwise unaffected. The slave widgets normally appear on top of the master but can be arranged to be underneath if necessary. This stacking order is defined by the order in which the widgets are created (this is discussed more in Section 21.8).
A geometry manager receives three pieces of information for use in computing a layout (see Figure 21.1). First, each slave widget requests a

particular width and height. These are usually the minimum dimensions the widget needs to display its information. For example, a button widget typically requests a size just large enough to display its text or image. Although geometry managers aren't obliged to satisfy the requests made by their slave widgets, they usually do.

Figure 21.1 A geometry manager determines the size and placement of slave widgets.


Requested sizes from slaves
Parameters from application designer
Geometry of master
Geometry manager
Sizes and locations of slaves
Requested sizes for master

The second kind of input for a geometry manager comes from the application designer and is used to control the layout algorithm. The nature of this information varies from geometry manager to geometry manager. For example, the gridder arranges slave widgets in rows and columns, so row and column numbers are needed to identify where to position the widgets.
The third piece of information used by geometry managers is the geometry of the master window. For example, the geometry manager might position a slave at the lower left corner of its master, or it might divide the space of the master among one or more slaves, or it might refuse to display a slave altogether if it doesn't fit within the area of its master.
After it has received all of the preceding information, the geometry manager executes a layout algorithm to determine the width, height, and position of each of its slaves. If the geometry manager assigns a slave a size different from what it requested, the widget must make do in the best way it can. Geometry managers usually try to give widgets the space they request, but they may produce more attractive layouts by giving widgets extra space in some situations. If there isn't enough space in a master for all of its slaves, some of the slaves may get less space than they requested. In extreme cases the geometry manager may choose not to display some slaves at all.

The controlling information for geometry management may change while an application runs. For example, a button might be reconfigured with a different font or bitmap, in which case it changes its requested dimensions. In addition, the geometry manager might be told to use a different approach (for example, arrange a collection of windows from top to bottom instead of left to right), some of the slave windows might be deleted, or the user might interactively resize the master window. When any of these things happens, the geometry manager recomputes the layout.

Some geometry managers set the requested size for the master window. For example, the gridder computes how much space is needed in the master to accommodate all of its slaves in the fashion requested by the application designer. It then sets the requested size for the master to these dimensions, overriding any request made by the master widget itself. This approach allows for hierarchical geometry management, where each master is itself the slave of another higher-level master. Size requests pass up through the hierarchy from each slave to its master, resulting ultimately in a size request for a toplevel window, which is passed to the window manager. Then actual geometry information passes down through the hierarchy, and the geometry manager at each level uses the geometry of a master to compute the geometry of one or more slaves. As a result, the entire hierarchy sizes itself to meet the needs of the lowest-level slaves; the overall effect is that the master widgets “shrink-wrap” around their slaves.

Each widget can be managed by at most one geometry manager at a time, although it is possible to switch geometry managers during the life of a slave. A widget can act as a master to any number of slaves, and a single geometry manager can simultaneously manage different groups of slaves associated with different masters. Although it is possible for different geometry managers to control different groups of slaves within the same master, it is not recommended.

Only internal widgets may be slaves for geometry management. The techniques described here do not apply to toplevel widgets since they are managed by the window manager for the display. See Chapter 26 for information on how to control the geometry of toplevel widgets.

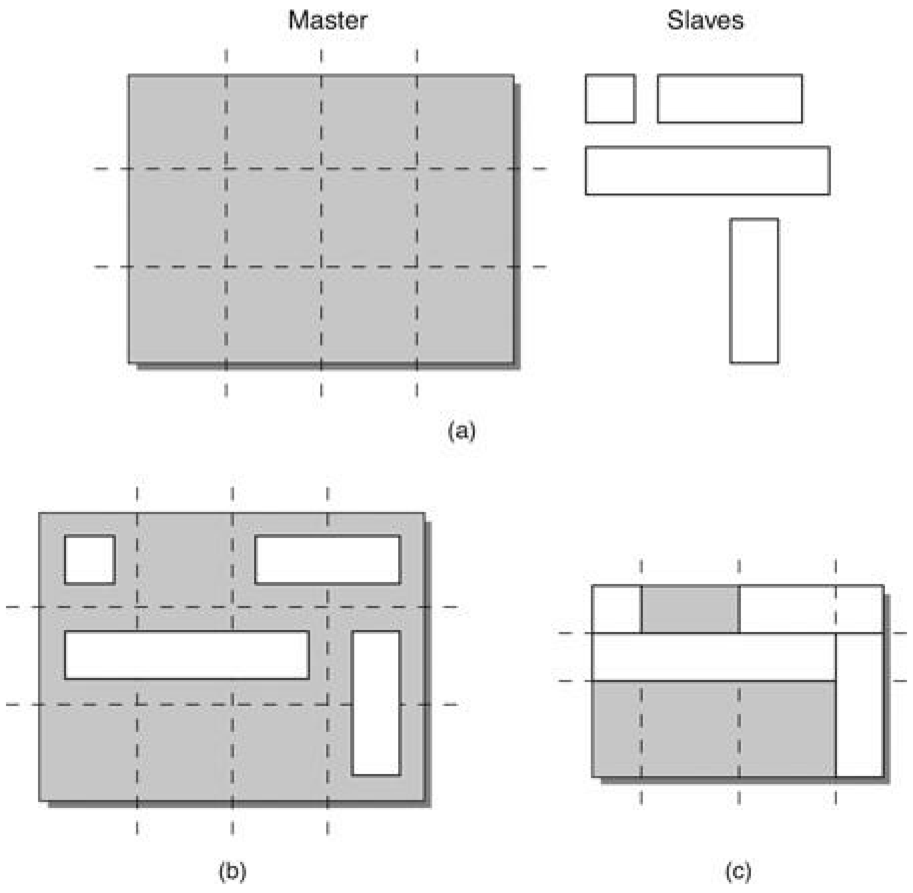
21.3 The Gridder

The gridder is a straightforward geometry manager that arranges slaves into rows and columns and allows them to span multiple rows and columns. Most application layout needs can be satisfied with the gridder options without resorting to additional frames and hierarchical layout techniques (see Section 21.7).

Each slave for a given master is assigned a row and column location, or *cell*, in the grid. The gridder shrink-wraps each column to the width of the largest slave in the column and correspondingly shrink-wraps each row to the height of the largest slave in the row, as shown in Figure 21.2. This is done in three steps:

1. The minimum size is determined for each row and column that accommodates all the slaves based on each slave's requested size. This minimum size then becomes the requested size for the master.
2. The actual size of the master is compared with this requested size, and then space is added to or removed from the layout as needed.
3. Each slave is positioned within its cell according to its configuration flags.

Figure 21.2 The gridder geometry manager arranges slave widgets into rows and columns.



The gridder geometry manager arranges slave widgets into rows and columns, as shown in Figure 21.2(a). Slaves can also span rows or columns to occupy more than one cell, as shown in Figure 21.2(b). The rows and columns “shrink-wrap” around the widgets so that the master is only as large as it needs to be to accommodate all the slaves, as illustrated in Figure 21.2(c).

The `grid configure` command (the word `configure` is optional, unless you are querying grid options) adds one or more widgets to grid control or changes the gridding options of widgets already under grid control:

```
grid slave ?slave ...? option value ?option value ...?
grid configure slave ?slave ...? ?option? \
    ?value option value ...?
```

There are a number of different configuration options for the gridder that control slave placement, stretching behavior, cell size, and grid size:

- `-column` *n*

Inserts the slave into the *n*th column.

- `-columnspan` *n*

Inserts the slave so that it occupies *n* columns (see Section 21.3.3).

- `-in` *master*

Use *master* as the master for the slave. *master* must be the slave’s parent or a descendant of the slave’s parent. If this option is not specified, the master defaults to the slave’s parent.

- `-ipadx` *distance*

Adds internal horizontal padding between the widget’s border and its content, expressed as a screen distance.

- `-ipady` *distance*

Adds internal vertical padding between the widget’s border and its content, expressed as a screen distance.

- `-padx` *distance*

Adds external horizontal padding around the widget, expressed as a screen distance; a two-element list can be provided to specify left and right padding independently (see Section 21.5).

- `-pady` *distance*

Adds external vertical padding around the widget, expressed as a screen distance; a two-element list can be provided to specify top and bottom padding independently (see Section 21.5).

- `-row` *n*

Inserts the slave into the *n*th row.

- `-rowspan` *n*

Inserts the slave so that it occupies *n* rows (see Section 21.3.3).

- `-sticky` *`style`*

Controls the position and stretching of a slave if a slave's cell is larger than its requested dimensions (see Section 21.3.1).
The gridder also supports `columnconfigure` and `rowconfigure` subcommands for setting and querying the configuration of columns and rows in the master:

```
grid columnconfigure master index ?option? ?option? \
    ?value option value ...?
grid rowconfigure master index ?option? ?option? \
    ?value option value ...?
```

If one or more options are provided, *`index`* may be given as a list of column indices to which the configuration options apply. Indices may be integers, names of widgets in the grid, or the keyword `all`. Currently supported options are

- `-minsize` *`amount`*

The minimum size for a column/row, expressed as a screen distance.

- `-pad` *`amount`*

Adds external padding around each widget in the column/row, expressed as a screen distance.

- `-weight` *`int`*

The relative weight for apportioning any extra space among columns when resizing (see Section 21.3.3).

- `-uniform` *`groupName`*

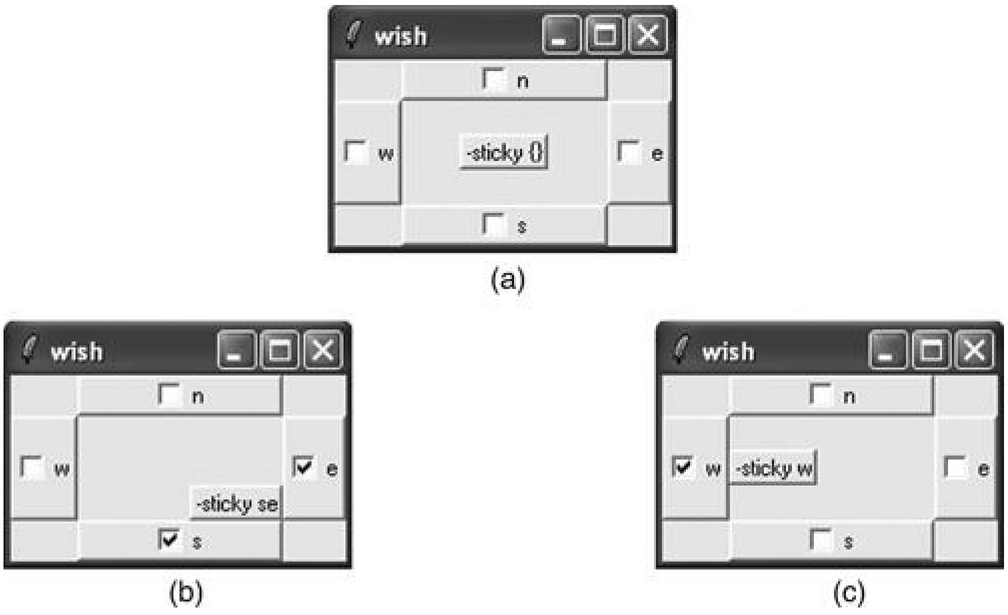
Symbolic name of a group of columns/rows whose resizing is proportionate (see Section 21.3.3).

21.3.1 The `grid` Command and the `-sticky` Options

The `grid` command is used to communicate with the gridder. In its simplest form the command takes a list of slaves and options and places each slave in a subsequent column starting with `0`. Each subsequent `grid` command for the same master starts a new row. The `-row` and `-column` options can be used to provide a specific cell or starting point.
If a slave does not entirely fill its cell, it is positioned in the center of its cell by default. The `-sticky` option is used to anchor or stretch a slave to any side or direction. This option takes any one or more of the characters `n`, `s`, `e`, or `w`. Each character causes the slave to stick to the corresponding side of the cell; if opposite sides are given, the slave is stretched between those two sides.

The following code demonstrates the effects of different `-sticky` options using a label widget surrounded by four checkbuttons in a 3-by-3 grid. Notice also that some cells, the four corners in this example, may be empty. In this case the master widget shows through. Figure 21.3 shows some examples from this application:

Figure 21.3 The label in the center demonstrates different `-sticky` options.



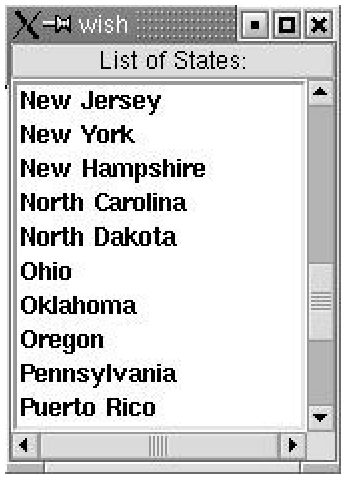
```
label .demo -textvariable stickyLabel -bd 2 -relief raised
checkbutton .n -text "n" -bd 2 -relief raised \
    -variable stickyN -onvalue n -offvalue {} \
    -command redo_sticky
checkbutton .s -text "s" -bd 2 -relief raised \
    -variable stickyS -onvalue s -offvalue {} \
    -command redo_sticky
checkbutton .e -text "e" -bd 2 -relief raised \
    -variable stickyE -onvalue e -offvalue {} \
    -command redo_sticky
checkbutton .w -text "w" -bd 2 -relief raised \
    -variable stickyW -onvalue w -offvalue {} \
    -command redo_sticky

grid .demo -row 1 -column 1
grid .n    -row 0 -column 1 -sticky nsew
grid .s    -row 2 -column 1 -sticky nsew
grid .e    -row 1 -column 2 -sticky nsew
grid .w    -row 1 -column 0 -sticky nsew

grid rowconfigure . 1 -weight 1
grid columnconfigure . 1 -weight 1

wm geometry . 180x100

proc redo_sticky {} {
    global stickyN stickyS stickyE stickyW stickyLabel

    append s2 $stickyN $stickyS $stickyE $stickyW
    grid .demo -row 1 -column 1 -sticky $s2
    set stickyLabel [list -sticky $s2]
}
```

21.3.2 Spanning Rows and Columns

A gridded widget can span multiple rows or columns. In the following code, the `-columnspan` option causes the `.label` widget to use both columns in the first row. This area is then treated as a single cell, as shown in Figure 21.4. Likewise, the `-rowspan` option can be used to span multiple rows.

```
grid .label -columnspan 2
grid .listbox -sticky nsew
grid .scrollbar -row 1 -column 1 -sticky ns
grid .xscrollbar -sticky ew
```

Figure 21.4 A label spanning multiple columns



List of States:

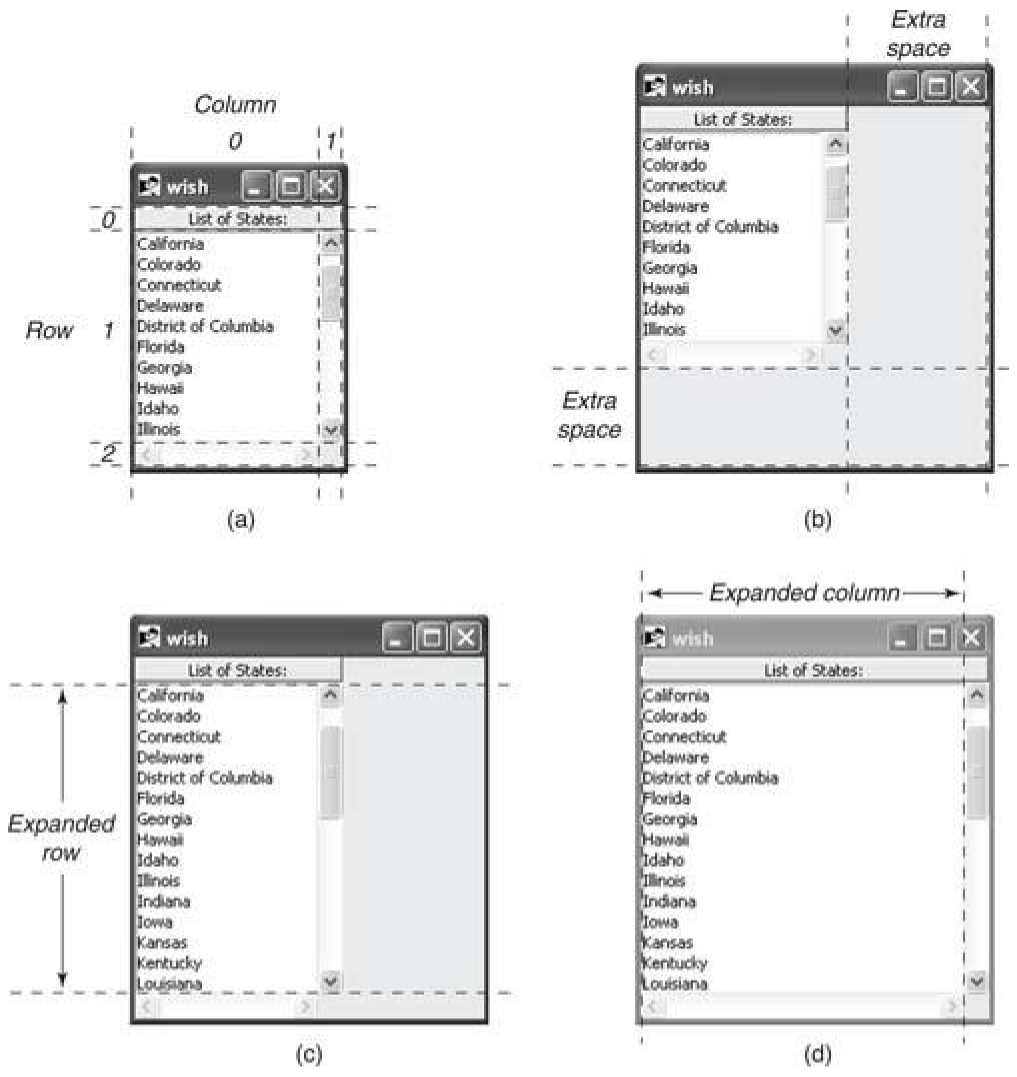
New Jersey
New York
New Hampshire
North Carolina
North Dakota
Ohio
Oklahoma
Oregon
Pennsylvania
Puerto Rico

When the gridder computes the minimum master size, it first looks at all the slaves with a row and column span of `1`. Then the slaves with row and column spans greater then `1` are examined, and if additional space is needed, the space is distributed across the spanned rows or columns.

21.3.3 Stretch Behavior and the `-weight` and `-uniform` Options

When a master window has more space than the slaves need, such as when a user interactively stretches a window, the gridder must allocate this extra space somewhere. There are several options that can control how the space is distributed. For example, in Figure 21.4 the desired behavior is for all additional space to be first given to the listbox widget, and then have the scrollbars stretch as needed. This can be accomplished easily by weighting the cell that the `.listbox` widget occupies, column 0 and row 1, with a higher *weight* than the others, as shown in Figure 21.5. Weights are assigned to rows and columns using the `-weight` option to the `grid rowconfigure` and `grid columnconfigure` commands. With the default weight of `0`, when the window shown in Figure 21.5(a) is resized as in Figure 21.5(b), the widgets do not expand to fill the space. When a weight of `1` is assigned to row 1, the rows expands to fill the space, as shown in Figure 21.5(c), and assigning a weight of `1` to column 0 expands the column to complete the expansion, shown in Figure 21.5(d).

Figure 21.5 Effect of using weighted rows and columns with the gridder



By default the gridder assigns a weight of `0` to all rows and columns. No additional space is allocated to rows or columns with a weight of `0`. Weights higher than `0` are allocated additional space proportionally, with `-weight 2` getting twice as much space as `-weight 1`.

The `-uniform` option is used to group rows and columns with similar weights. This way a set of rows or columns can be configured to always have the same size. For example, multiple rows configured with `-weight 1 -uniform a` all have the same height. See the reference documentation for more information.

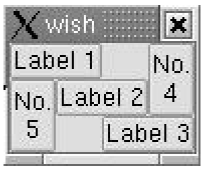
21.3.4 Relative Placement Characters

The `grid` command provides a shorthand notation for placing slaves without having to specify explicit row and column numbers. By default, each slave specified in a `grid` command is placed in subsequent columns, and each new

`grid` command starts a new row. This creates an organization in the source code that represents the layout on the screen.
There are three relative placement characters that can be used where a slave would normally appear in the `grid` command: `-`, `x`, and `^`. The `-` character extends the previous slave into the next column, and each subsequent `-` character continues to increase the column span for the slave. The `x` character leaves the cell blank. The `^` character extends the slave in the previous row down into the current cell, and each `^` in subsequent `grid` commands continues to increase the row span for the slave. Figure 21.6 demonstrates the use of each of these relative placement characters using the following code:

```
ttk::label .w1 -text "Label 1" -relief raised
ttk::label .w2 -text "Label 2" -relief raised
ttk::label .w3 -text "Label 3" -relief raised
ttk::label .w4 -text "No.\n4"  -relief raised
ttk::label .w5 -text "No.\n5"  -relief raised

grid   .w1   -    x    .w4   -sticky nsew
grid   .w5  .w2   -     ^    -sticky nsew
grid    ^    x   .w3    -    -sticky nsew
```

Figure 21.6 Relative placement characters can simplify grid layout commands.

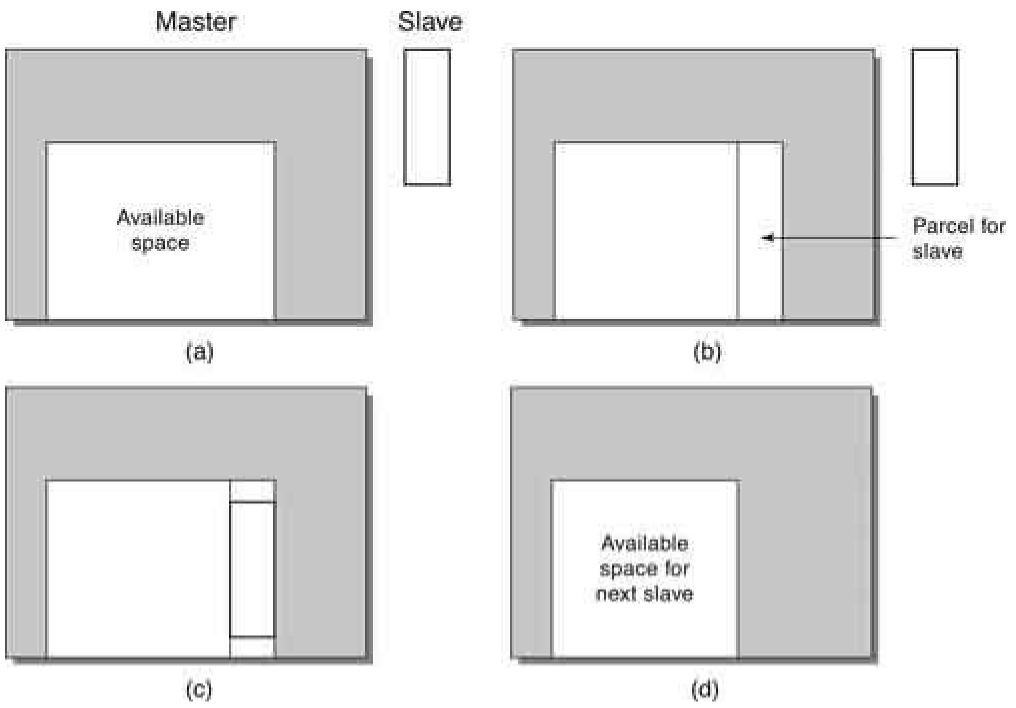


21.4 The Packer

The packer arranges the slaves for a master by positioning them one at a time in the master's window, working from the edges toward the center. The packer is easy to use in simple cases where a single row or column is needed, but it is also capable of complex layouts. Although you can achieve any arrangement of widgets with either the packer or the gridder, complex layouts are usually easier to create with the gridder.
The packer maintains a list of slaves for a given master window, called the

packing list. The packer arranges the slaves by processing the packing list in order, packing one slave in each step. At the time a particular slave is processed, part of the area of the master window has already been allocated to earlier slaves on the list, leaving a rectangular unallocated area for all the remaining slaves, called the *cavity*, as shown in Figure 21.7(a). The current slave is positioned in three steps: allocate a parcel, stretch the slave, and position it in the parcel.

Figure 21.7 The packer arranges slaves around the edges of the master.



In the first step a rectangular region called a *parcel*, or *packing space*, is allocated from the available space. This is done by "slicing" off a piece along one side of the available space. For example, in Figure 21.7(b) a parcel is sliced from the right side of the available space. One dimension of the parcel is determined by the size of the available space, and the other dimension is controllable. The controllable dimension is normally taken from the slave's requested size in that dimension, but the packer allows you to request additional space if you wish. In Figure 21.7(b) the width is the controllable dimension; if the parcel had been sliced from the top or bottom, the parcel's height would have been controllable instead of its width.
In the second step the packer chooses the dimensions of the slave. By default the slave gets the size it requested, but you can specify instead that it should be stretched in one or both dimensions to fill the space of the parcel.

If the slave's requested size is larger than the parcel, it is reduced to fit the size of the parcel.
The third step is to position the slave inside its parcel. If the slave is smaller than the parcel, an anchor position may be given for the slave, such as `n`, `s`, or `center`. In Figure 21.7(c) the slave has been positioned in the center of the parcel, which is the default.
Once the slave has been positioned, a smaller rectangular region is left for the next slave to use, as shown in Figure 21.7(d). If a slave doesn't use all of the space in its parcel, as in Figure 21.7, the leftover space is not used for later slaves. Thus each step in the packing starts with a rectangular region of available space and ends up with a smaller rectangular region.
Once all of the slaves have been packed, the packer then sizes the master widget to meet the needs of the lowest-level slaves, with the overall effect that the master widget "shrink-wraps" around its slaves.
The `pack configure` command (the word `configure` is optional, unless you are querying packer options) adds one or more widgets to packer control or changes the packing options of widgets already under packer control:

```
pack slave ?slave ...? option value ?option value ...?
pack configure slave ?slave ...? ?option? \
    ?value option value ...?
```

There are a number of different configuration options for the packer that control slave characteristics such as placement, stretching behavior, cell size, and positioning:

- `-after` *`widget`*

Use *`widget`*'s master as the master for the slave, and insert the slave into the packing list just after *`widget`*.

- `-anchor` *`position`*

Controls the position of a slave if a slave's cell is larger than its requested dimensions (see Section 21.4.4). Defaults to `center`.

- `-before` *`window`*

Use *`window`*'s master as the master for the slave and insert the slave into the packing list just before *`window`*.

- `-expand` *`boolean`*

If *`boolean`* is a true value, the slave's parcel grows to absorb any extra space left over in the master (see Section 21.4.3). Defaults to false.

- `-fill` *`style`*

Specifies whether (and how) to grow the slave if its parcel is larger than the slave's requested size (see Section 21.4.2). *`style`* must be either `none`, `x`, `y`, or `both`. Defaults to `none`.

- `-in` *`master`*

Use *`master`* as the master for the slave. *`master`* must be the slave's parent or a descendant of the slave's parent. If this option is not specified, the master defaults to the slave's parent.

- `-ipadx` *`distance`*

Specifies internal horizontal padding added between the widget's border and its content, expressed as a screen distance.

- `-ipady` *`distance`*

Specifies internal vertical padding added between the widget's border and its content, expressed as a screen distance.

- `-padx` *`distance`*

Specifies external horizontal padding added around the widget, expressed as a screen distance; a two-element list can be provided to specify left and right padding independently (see Section 21.5).

- `-pady` *`distance`*

Specifies external vertical padding added around the widget, expressed as a screen distance; a two-element list can be provided to specify top and bottom padding independently (see Section 21.5).

- `-side` *`side`*

`side` specifies which side of the master the slave should be packed against (see Section 21.4.1). Must be `top`, `bottom`, `left`, or `right`. Defaults to `top`.

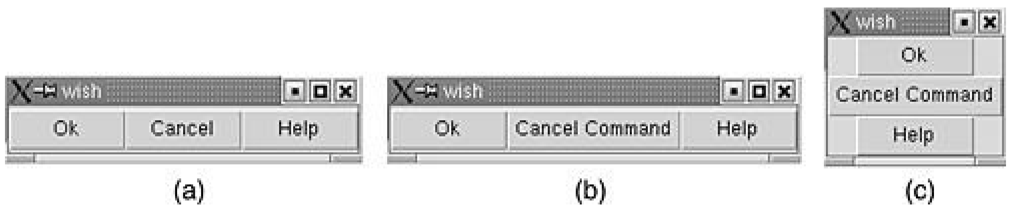
21.4.1 The `pack` Command and `-side` Options

The `pack` command is used to communicate with the packer. In its simplest form it takes one or more widget names as arguments, followed by one or more pairs of additional arguments that indicate how to manage the widgets. For example, consider the following script, which creates three buttons and arranges them in a row:

```
ttk::button .ok -text OK
ttk::button .cancel -text Cancel
ttk::button .help -text Help
pack .ok .cancel .help -side left
```

The `pack` command asks the packer to manage `.ok`, `.cancel`, and `.help` as slaves and to pack them in that order. The master for the slaves defaults to their parent, which is ., the main widget. The option `-side left` applies to all three of the widgets; it indicates that the parcel for each slave should be allocated on the left side of the available space. Because no other options are specified, the parcel for each slave is allocated to be just large enough for the slave. This causes the slaves to be arranged in a row from left to right across the master. Furthermore, the packer computes the minimum size needed by the master to accommodate all of its slaves, and it sets the master's requested size to these dimensions. The window manager sets the main widget's size as requested, so the master ends up shrink-wrapped around the slaves, as shown in Figure 21.8(a).

Figure 21.8 Simple packer examples



The packer recomputes the layout whenever any of the relevant information changes. For example, if you type the command

```
.cancel configure -text "Cancel Command"
```

to change the text in the middle button, the button changes its requested size and the packer changes the layout to make more space for it, as in Figure 21.8(b). If you then type the command

```
pack .ok .cancel .help -side top
```

the layout changes so that each slave is allocated at the top of the remaining space, producing the columnar arrangement shown in Figure 21.8(c). In this case, the master's width is chosen to accommodate the largest of the slaves (`.cancel`). Each of the other slaves receives a parcel wider than it needs, and the slave is centered in its parcel.

21.4.2 Filling

If a parcel ends up larger than the size its slave requested, you can use the `-fill` option to stretch the slave so that it fills the parcel in one or both directions. For example, consider the column of widgets in Figure 21.8(c): each widget is given its requested size, which results in a somewhat ragged look because the buttons are all different sizes. The `-fill` option can be used to make all the buttons the same size:

```
pack .ok .cancel .help -side top -fill x
```

Figure 21.9 shows the effect produced by this command: each of the slaves is stretched horizontally to fill its parcel, and since the parcels are as wide as the master window, all of the slaves end up with the same width.

Figure 21.9 The effect of using `-fill`

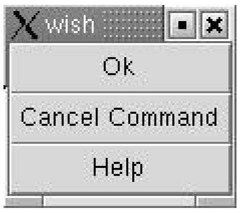
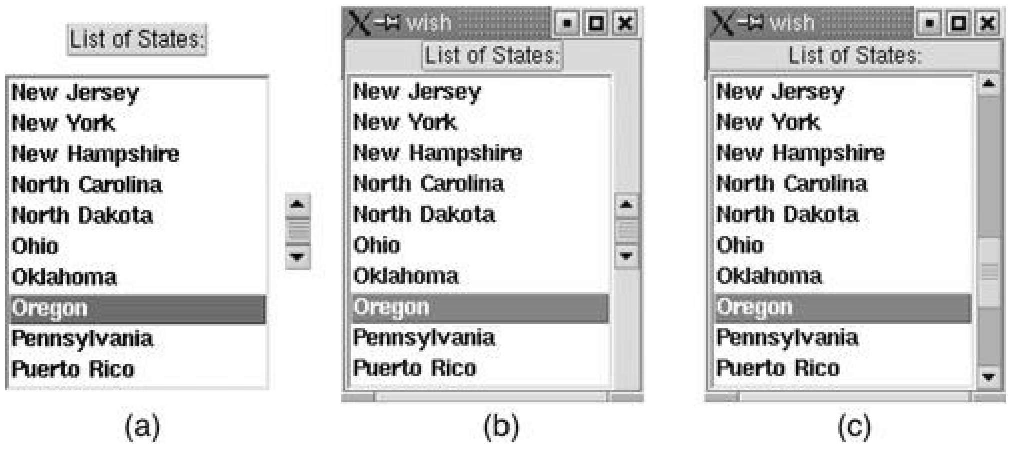


Figure 21.10 shows another simple example of filling. The three windows are configured differently, so a separate `pack` command is used for each one. The order of the `pack` commands determines the order of the windows in the packing list:

```
pack .label -side top -fill x
pack .scrollbar -side right -fill y
pack .listbox
```

Figure 21.10 Another packer example using the `-fill` and `-side` options



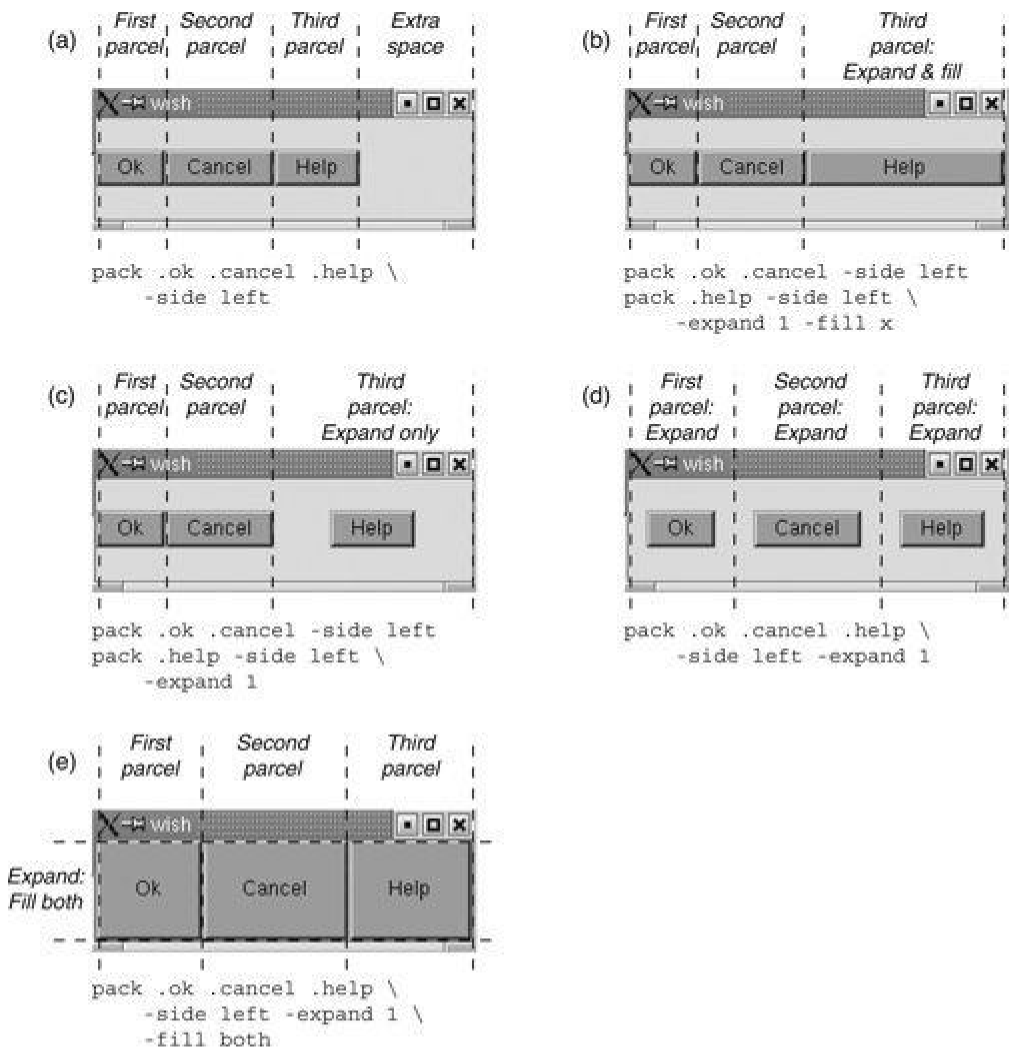
The `.label` widget is packed first, and it occupies the top part of the master window. The `-fill x` option specifies that the window should be stretched horizontally so that it fills its parcel. The scrollbar widget is packed next, in

a similar fashion except that it is arranged against the right side of the window and stretched vertically. The widget `.listbox` is packed last. No options need to be specified for it: it ends up in the same place regardless of which side it is packed against.
Without the `-fill` option, the label and scrollbar would simply occupy the center of the parcel, as Figure 21.10(b) illustrates. Adding `-fill` causes them to stretch and occupy the full parcel, as seen in Figure 21.10(c).

21.4.3 Expansion

Sometimes a master window has more space than its slaves need. This can occur, for example, if a user interactively stretches a window. When this happens, the default behavior of the packer is to leave the extra space unused, as in Figure 21.11(a). However, you can use the `-expand` option to tell the packer to give any extra space to a particular slave (for example, expand a text widget to fill all the extra space), or to divide the extra space uniformly among a collection of slaves (for example, distribute a collection of buttons uniformly across the available space). The `-expand` option makes it possible to produce layouts that are attractive regardless of how the user resizes the window, so that the user can choose the size he or she prefers. In each of the examples in Figure 21.11, the size of the main widget was fixed with the command `wm geometry . 232x62` to simulate what would happen if the user had interactively resized the window.

Figure 21.11 Examples of using `-expand` and `-fill`



(a)
```
pack .ok .cancel .help \
    -side left
```

(b)
```
pack .ok .cancel -side left
pack .help -side left \
    -expand 1 -fill x
```

(c)
```
pack .ok .cancel -side left
pack .help -side left \
    -expand 1
```

(d)
```
pack .ok .cancel .help \
    -side left -expand 1
```

(e)
```
pack .ok .cancel .help \
    -side left -expand 1 \
    -fill both
```

Figure 21.11 shows how `-expand` works. If you specify `-expand` for a slave, as with `.help` in Figure 21.11(b), the slave's parcel expands to include any extra space in the master. In Figure 21.11(b), `-fill x` has also been specified for `.help`, so the slave window is stretched to cover the entire width of the parcel. Figure 21.11(c) is similar to Figure 21.11(b) except that no filling is specified, so `.help` is simply centered in its parcel.

If you specify `-expand` for multiple slaves, their parcels share the extra space equally. Figure 21.11(d) shows what happens if all of the windows are packed with `-expan`d but no filling, and Figure 21.11(e) shows what happens when the slaves are also filled in both dimensions.

Note

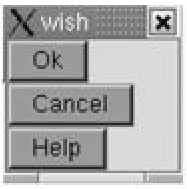
The options `-expand` and `-fill` are often confused because of the similarity of their names. The `-expand` option determines whether a parcel absorbs extra space in the master, and `-fill` determines whether a slave's window absorbs extra space in its parcel. The options are often used together so that a slave's window absorbs all the extra space in its master.

21.4.4 Anchors

If a parcel has more space than its slave requested (for example, because you specified `-expand` for it), and if you choose not to stretch the slave with the `-fill` option, the packer normally centers the slave in its parcel. The `-anchor` option allows you to request a different position: its value specifies where the slave should be positioned in the parcel, in one of the forms listed in Section 18.15.3. For example, `-anchor nw` specifies that the slave should be positioned in the upper left corner of its parcel. Figure 21.12(a) shows how `-anchor` can be used to left-justify a collection of buttons.

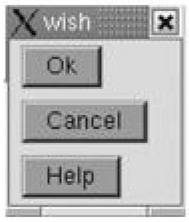
Figure 21.12 Examples of the `-anchor` option

(a)



```
pack .ok .cancel .help -side left
```

(b)



```
pack .ok .cancel -side left
pack .help -side left -expand 1 -fill x
```

The `-fill` and `-anchor` options preserve external padding requested with `-padx` and `-pady`. For example, Figure 21.12(b) is the same as Figure 21.12(a) except that external padding has been requested. In this case the buttons aren't positioned at the left edges of their parcels, but instead they are inset by the `-padx` distance.

21.4.5 Packing Order

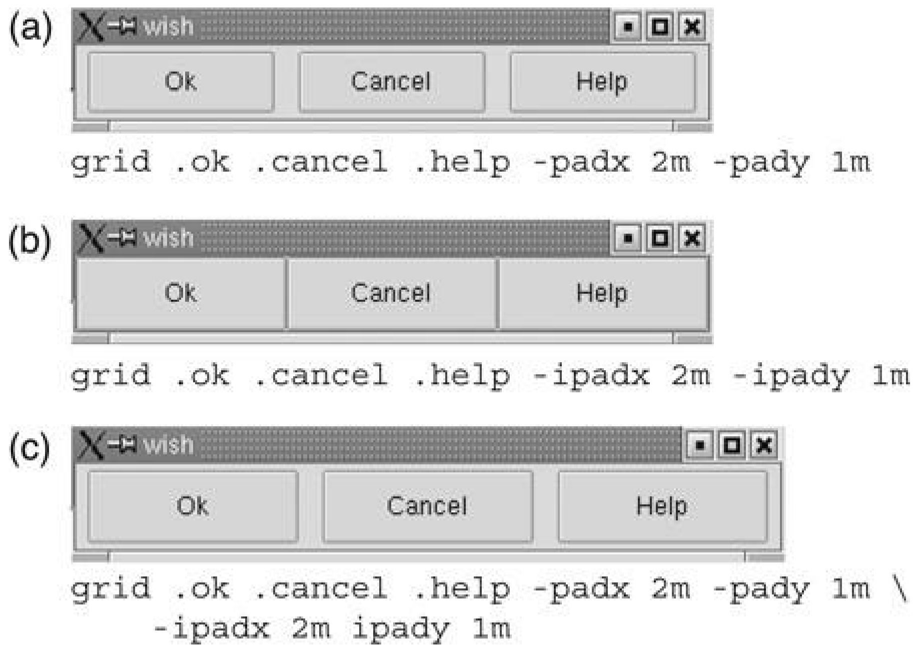
The `-before` and `-after` options allow you to control a slave's position in the packing list for its master. If you don't specify one of these options, each new slave goes at the end of the packing list. If you specify `-before` or `-after`, its value must be the name of a window that is already packed, and the new slave(s) are positioned just before or after the given slave, respectively.

21.5 Padding

Both the packer and the gridder provide four options for requesting extra space for a slave. The extra space is called *padding*, and it comes in two forms: *external padding* and *internal padding*. External padding is requested with the `-padx` and `-pady` options; these options cause the packer to allocate a parcel larger than that requested by the slave and leave extra space around the outside of the slave. For example, the widgets in Figure 21.13(a) are packed with the options

```
-padx 2m -pady 1m
```

Figure 21.13 Examples of padding



(a)

```
grid .ok .cancel .help -padx 2m -pady 1m
```

(b)

```
grid .ok .cancel .help -ipadx 2m -ipady 1m
```

(c)

```
grid .ok .cancel .help -padx 2m -pady 1m \
     -ipadx 2m ipady 1m
```

These options specify that there should be 2 millimeters of extra space on each side of each of the widgets and 1 millimeter of extra space above and below each widget. Internal padding is requested with the `-ipadx` and `-ipady` options; they also cause a parcel to be larger than what the slave requested, but in this case the slave window is enlarged to incorporate the extra space, as in Figure 21.13(b). External and internal padding can be used together, as in Figure 21.13(c), and different slaves can have different amounts of padding.

21.6 The Placer

The placer is the most simplistic geometry manager and is used typically in specialized applications. Slave widgets are placed by the manager precisely where specified by the `place` command options; there are no special algorithms applied to the layout. It is simplistic because it does very little work, putting the onus on the application to manage the layout. The placer should be used only in situations where the gridder or packer cannot perform the required layout. The placer does provide a relative placement option, which gives a "rubber sheet" effect to the slave widgets. Slave windows managed by the placer do not affect other slave geometries as is the case in other geometry managers, nor is size information propagated to the master.

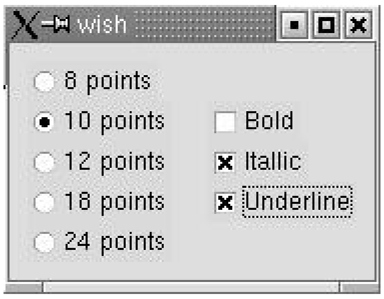
The `place` command is used to communicate with the placer. In its simplest form it takes one or more widget names as arguments, followed by one or more pairs of additional arguments that indicate how to place the widgets. Placement is controlled by an exact location (`-x`, `-y`) and by an exact size (`-width`, `-height`) and is a position within the master window, or relative placement may be used where the location (`-relx`, `-rely`) and size (`-relwidth`, `-relheight`) are specified as floating-point numbers relative to the master. For the other options available for the placer, see the reference documentation.

21.7 Hierarchical Geometry Management

Both the packer and the gridder can be used in hierarchical arrangements where slave windows are also masters for other slaves. Figure 21.14 shows an example of hierarchical packing. The resulting layout contains a column of radiobuttons on the left and a column of checkbuttons on the right, and each group of buttons is centered vertically in its column. To achieve this effect, two extra frame widgets, `.left` and `.right`, are packed side by side in the main window, then the buttons are packed inside them. The packer sets the requested sizes for `.left` and `.right` to provide enough space for the buttons, then uses this information to set the requested size for the main widget. The main widget's geometry is set to the requested size by the window manager, then the packer arranges `.left` and `.right` inside it, and finally it arranges the buttons inside `.left` and `.right`. Here is the code that produced the layout:

```
frame .left
frame .right
set buttonList [list]
foreach size {8 10 12 18 24} {
    ttk::radiobutton .pts$size -text "$size points" \
        -variable pts -value $size
    lappend buttonList .pts$size
}
ttk::checkbutton .bold -text Bold -variable bold
ttk::checkbutton .italic -text Italic -variable italic
ttk::checkbutton .underline -text Underline \
    -variable underline
pack {*}$buttonList -in .left -side top -anchor w
pack .bold .italic .underline -in .right -side top \
    -anchor w
pack .left -side left -padx 3m -pady 3m
pack .right -side right -padx 3m -pady 3m
```

Figure 21.14 An example of hierarchical packing



This code also illustrates a situation where a slave's master is different from its parent (by using the `-in` option to specify a master). It would have been possible to create the button windows as children of `.left` and `.right` (for example, `.left.pts8` instead of `.pts8`), but it is cleaner to create them as children of . and then pack them inside `.left` and `.right`. The windows `.left` and `.right` serve no purpose in the application except to help in geometry management; they have the same background color as the main widget, so they are not even visible on the screen. If the buttons were children of their geometry masters, changes to the geometry management (such as adding more levels in the packing hierarchy) might require the button windows to be renamed and would break any code that used the old names. It is better to give windows names that reflect their logical purpose in the application, build separate frame hierarchies where needed for geometry management, and then pack the functional windows into the frames.

Note

In the example in Figure 21.14, the frames `.left` and `.right` must be created before the buttons so that the buttons are stacked on top of the frames. The most recently created widget is highest in the stacking order; if `.left` is created after `.pts8`, it is on top of `.pts8`, so `.pts8` is not visible. Alternatively, the `raise` and `lower` commands can be used to adjust the stacking order (see Section 21.8).

A slave's master must be either its parent or a descendant of its parent. The reason for this restriction has to do with the clipping rules in some windowing systems. Each window is clipped to the boundaries of its parent, so no portion of a child that lies outside of its parent is displayed. Tk's restriction on master windows guarantees that a slave is visible and unclipped if its master is visible and unclipped. Suppose that the restriction were not enforced, so that window `.x.y` could have `.a` as its master. Suppose also that `.a` and `.x` do not overlap at all. If you asked the packer to position `.x.y` in `.a`, the packer would set `.x.y`'s position as requested, but this would cause `.x.y` to be outside the area of `.x`, so the windowing system would not display it even though `.a` is fully visible. This behavior would be confusing to application designers, so Tk restricts mastership to keep the behavior from occurring. The restriction applies to all of Tk's geometry managers.

21.8 Widget Stacking Order

Stacking order determines the layering of widgets on the screen. If two sibling widgets overlap, the stacking order determines which one appears on top of the other. Widgets are normally stacked in the order of their creation: each new widget goes at the top of the stacking order, obscuring any of its siblings that were created before it. The `raise` and `lower` commands allow you to change the stacking order of widgets:

```
raise .w
raise .w .x
lower .w
lower .w .x
```

The first command raises `.w` so that it is at the top of the stacking order (it obscures all of its siblings). The second places `.w` just above `.x` in the stacking order. The third command lowers `.w` to the bottom of the stacking order (it is obscured by all of its siblings), and the last command places `.w` just below `.x` in the stacking order.
Most of the time the stacking order is not important, since most widgets are positioned without overlap. When the `place` window manager is used, widgets can overlap, and when the `-in` option is used with any of the window managers, the slave widget must be above the master; otherwise the slave widget is not visible.

Note

You can use `raise` and `lower` for toplevel widgets as well as internal widgets, but this does not work for all window managers. Some window managers prevent applications from raising and lowering their toplevel windows; with these window managers you must raise or lower toplevel widgets manually using window manager functions.

21.9 Other Geometry Manager Options

So far the geometry manager commands have been discussed in their most common form, where the first arguments are the names of slave windows and the last arguments specify configuration options. Section 21.1 shows several other common forms for the geometry manager commands, where the first argument selects one of several operations. These commands apply to all three geometry managers: `grid`, `pack`, and `place`. The `grid configure` command, for example, has the same effect as the short form that's been used up until now; the remaining arguments specify windows and configuration options. If, for example, `grid configure` (or the short form with no command option) is applied to a window that is already managed by the

gridder, the slave's configuration is modified; configuration options not specified in the `grid` command retain their old values.
The command option `slaves` returns a list of all of the slaves for a given master window. For the packer, the order of the slaves reflects their order in the packing list:

```
  pack slaves .left
⇒.pts8 .pts10 .pts12 .pts18 .pts24
```

The command option `info` returns all of the configuration options for a given slave:

```
  pack info .pts8
⇒ -in .left -anchor w -expand 0 -fill none -ipadx 0?-ipady 0
  -padx 0 -pady 0 -side top
```

The return value is a list consisting of names and values for configuration options in exactly the form in which you would specify them to `pack configure`. This command can be used to save the state of a slave so that it can be restored later.
The command option `forget` causes the geometry manager to stop managing one or more slaves and forget all of its configuration state for them. It also unmaps the windows so that they no longer appear on the screen. This command can be used to transfer control of a window from one geometry manager to another, or simply to remove a window from the screen. If a forgotten window is itself a master for other slaves, the information about those slaves is retained, but the slaves won't be displayed on the screen until the master window becomes managed again.

Note

The gridder is unique in having a `remove` command option. It removes slaves from the grid and unmaps their windows, just like the `forget` command option. However, the configuration options for that window are remembered, so that if the slave is managed once more by the gridder, the previous values are retained.

The command option `propagate` allows you to control whether or not the packer or gridder sets the requested size for a master window. Normally these geometry managers set the requested size for each master window to just accommodate the needs of its slaves, and it updates the requested size as the needs of the slaves change. This feature is called *geometry propagation*, and it overrides any size that the master might have requested for itself. You can disable propagation with one of the following commands, depending on which geometry manager is controlling the layout of the master:

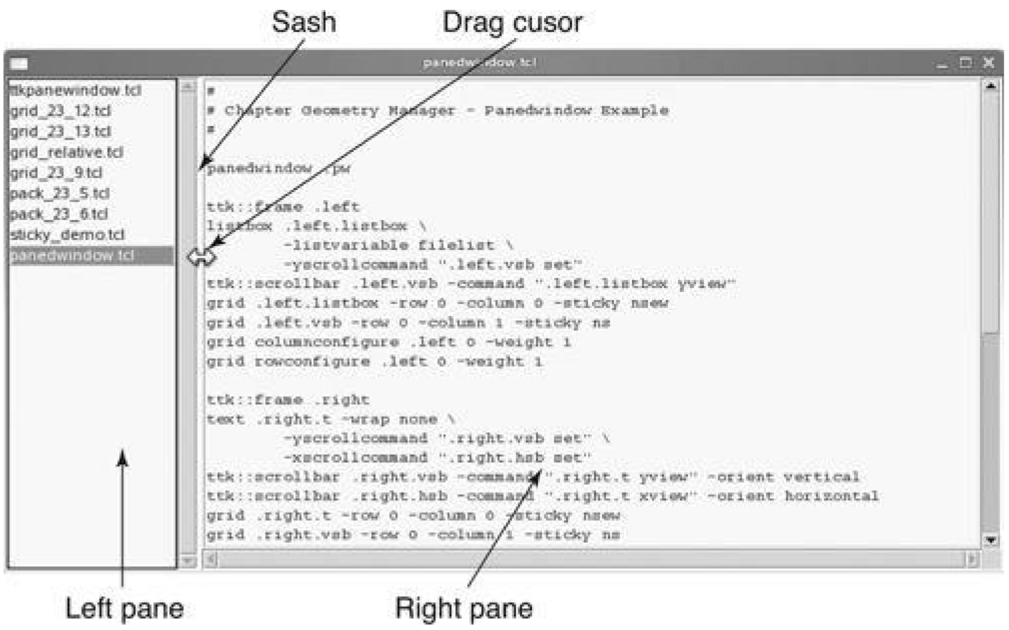
```
pack propagate master 0
grid propagate master 0
```

where `master` is the name of the master window. This instructs Tk not to set the requested size for `master`, so that the size requested by the master widget itself is used. You can resume propagation by providing a value of `1` to the command instead of `0`.
Each geometry manager has additional options unique to its algorithm. See the reference documentation for a complete listing.

21.10 Other Geometry Managers in Tk

As mentioned earlier, there are Tk widgets that can act as geometry managers; that is, they can map widgets onto the display and control their size and placement. The text widget is capable of embedding windows into a text document where each window is treated like a font glyph. The requested size of the window becomes the glyph size, and the text widget maps the window accordingly. The canvas widget can also embed windows and map them to the display. It does this according to its own method of placing objects onto the canvas. You can find out more information about the canvas and text widgets in Chapter 23 and Chapter 24, respectively.
The panedwindow (and its themed counterpart, ttk::panedwindow) takes one or more widgets and displays them in a horizontal row or vertical column. The widgets are separated by a sash that the user can manipulate with the mouse. When the sash is moved, each widget on either side of the sash is resized. Figure 21.15 illustrates a panedwindow with two panes. The left pane contains a frame with a listbox and a scrollbar. The right pane contains a frame with a text widget and scrollbars. The sash width and the handle size are fully configurable.

Figure 21.15 The panedwindow widget has features of a geometry manager.

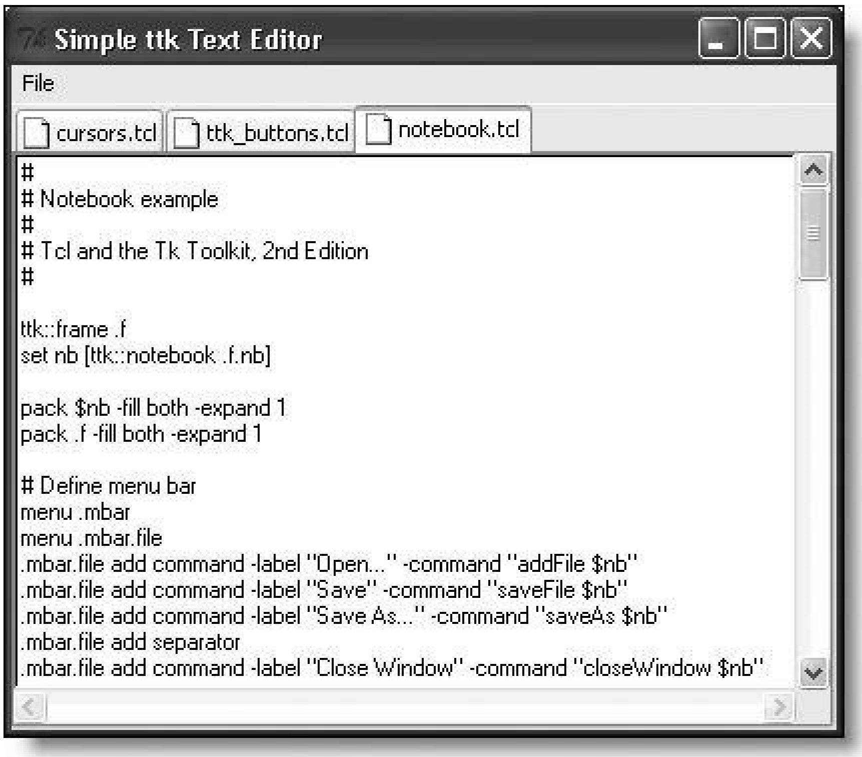


```
ttkpanewindow.tcl
grid_23_12.tcl
grid_23_13.tcl
grid_relative.tcl
grid_23_9.tcl
pack_23_5.tcl
pack_23_6.tcl
sticky_demo.tcl
panedwindow.tcl
```

```
#
# Chapter Geometry Manager - Panedwindow Example
#

panedwindow .pw

ttk::frame .left
listbox .left.listbox \
        -listvariable filelist \
        -yscrollcommand ".left.vsb set"
ttk::scrollbar .left.vsb -command ".left.listbox yview"
grid .left.listbox -row 0 -column 0 -sticky nsew
grid .left.vsb -row 0 -column 1 -sticky ns
grid columnconfigure .left 0 -weight 1
grid rowconfigure .left 0 -weight 1

ttk::frame .right
text .right.t -wrap none \
        -yscrollcommand ".right.vsb set" \
        -xscrollcommand ".right.hsb set"
ttk::scrollbar .right.vsb -command ".right.t yview" -orient vertical
ttk::scrollbar .right.hsb -command ".right.t xview" -orient horizontal
grid .right.t -row 0 -column 0 -sticky nsew
grid .right.vsb -row 0 -column 1 -sticky ns
```

Similarly, the themed notebook widget, ttk::notebook, serves in part as a geometry manager. When you add more widgets for it to manage, it displays a set of tabs corresponding to those slaves. All the slaves are unmapped except for the one whose tab is selected. Selecting a different tab unmaps the current slave and maps the slave of the tab selection. Figure 21.16 shows an example of a notebook with three tabs. The selected tab has mapped a slave frame containing a gridded text widget and vertical and horizontal scrollbars.

Figure 21.16 The notebook widget has features of a geometry manager.



```
#
# Notebook example
#
# Tcl and the Tk Toolkit, 2nd Edition
#

ttk::frame .f
set nb [ttk::notebook .f.nb]

pack $nb -fill both -expand 1
pack .f -fill both -expand 1

# Define menu bar
menu .mbar
menu .mbar.file
.mbar.file add command -label "Open..." -command "addFile $nb"
.mbar.file add command -label "Save" -command "saveFile $nb"
.mbar.file add command -label "Save As..." -command "saveAs $nb"
.mbar.file add separator
.mbar.file add command -label "Close Window" -command "closeWindow $nb"
```

Finally, the `wm` command is used to manage toplevel windows, typically windows created using the `toplevel` command, but it can also be used to “dock” (stop managing a toplevel window) or “undock” (make into a toplevel window) any widget. See Chapter 26 for more details on the `wm` command.

22. Events and Bindings

You have already seen that Tcl scripts can be associated with widgets such as buttons or menus so that the scripts are invoked when certain events occur, such as clicking a mouse button over the widget. These mechanisms are provided as specific features of specific widget classes. Tk also provides a more general mechanism called *bindings*, which allow you to create handlers for any *event* in any window. A binding "binds" a Tcl script to an event or sequence of events in one or more windows so that the script is evaluated whenever the given event sequence occurs in any of the windows. You can use bindings to extend the basic functions of a widget, for example, by defining shortcut keys for common actions. You can also override or modify the default behaviors of widgets, since they are implemented with bindings.

22.1 Commands Presented in This Chapter

This chapter discusses the following commands for manipulating events and bindings:

- `bind` *`tag sequence script`*

Arranges for *`script`* to be evaluated each time the event sequence given by *`sequence`* occurs in the window(s) given by *`tag`*. If a binding already exists for *`tag`* and *`sequence`*, it is replaced. If *`script`* is an empty string, the binding for *`tag`* and *`sequence`* is removed, if there is one. If the first character of *`script`* is `+`, the script is appended to any existing script for the binding.

- `bind` *`tag sequence`*

If there is a binding for *`tag`* and *`sequence`*, returns its script. Otherwise it returns an empty string.

- `bind` *`tag`*

Returns a list of the sequences for which *`tag`* has bindings.

- `bindtags` *`widget`* `?`*`tagList`*`?`

Sets the bindings tags associated with *`widget`* to *`tagList`*. Returns a list of binding tags currently associated with *`widget`* if *`tagList`* is omitted.

- `event add <<`*`virtual`*`>>` *`sequence`* `?`*`sequence`*`...?`

Associates the virtual event *`virtual`* with the physical event *`sequence`*(s).

- `event delete <<`*`virtual`*`>> ?` *`sequence sequence`*`...?`

Deletes each `sequence` from those associated with the virtual event given by `virtual`. If no `sequence` is given, all sequences are removed.

- `event generate` *`widget event ?option value option value ...?`*

Generates a window event and arranges for it to be processed just as if it had come from the window system. `widget` gives the path name of the window for which the event is generated. `event` may have any of the forms allowed for the sequence argument of the `bind` command except that it must consist of a single event pattern, not a sequence. The `option value` pairs given are used in the `%` substitutions for the event binding.

- `event info ?<<`*`virtual`*`>>?`

Returns sequences associated with a specified virtual event or a list of currently defined virtual events if no event is specified.

22.2 Events

An *event* is a record generated by the windowing system to notify an application of a potentially interesting occurrence. Each event has a *type* that indicates the general sort of thing that occurred. The event types that are most commonly used in bindings are those for user actions such as key presses or changes in the pointer position. There are many other event types, such as those generated when windows change size, when windows are destroyed, when windows need to be redisplayed, and so on. The following are the most commonly used event types, but see the reference documentation for the `bind` command for details:

- `Key` or `KeyPress`—A key was pressed.
- `KeyRelease`—A key was released.
- `Button` or `ButtonPress`—A mouse button was pressed.
- `ButtonRelease`—A mouse button was released.
- `Enter`—The mouse pointer moved into a widget (it is now over a visible portion of the widget).
- `Leave`—The mouse pointer moved out of a widget.
- `Motion`—The mouse pointer moved from one point to another within a single widget.
- `MouseWheel`—The user moved the mouse wheel.
- `FocusIn`—A widget received keyboard focus.
- `FocusOut`—A widget lost keyboard focus.
- `Configure`—A widget was initially displayed or changed its size, position, or border width.
- `Map`—A widget became viewable.

- `Unmap`—A widget is no longer viewable.
- `Destroy`—A widget has been destroyed.

Note

Commands that reference event types and their modifiers are case-sensitive. Make sure that you have the exact capitalization for the event type and its modifiers, or the command will raise an error.

In addition to its type, each event also contains several other fields. For mouse events, one of them specifies a widget and two others give the x- and y-coordinates of the pointer within the widget. For `<Enter>` and `<Leave>` events, the widget is the one just entered or left. For `<ButtonPress>`, `<ButtonRelease>`, and `<Motion>` events, the widget is the one currently under the pointer. For `<KeyPress>` and `<KeyRelease>` events, the widget is the one that has the application's *input focus* (see Chapter 27 for details on the input focus).
Button and key events also contain a field called the *detail*, which indicates the particular button or key that was pressed. For `<ButtonPress>` and `<ButtonRelease>` events, the detail is a button number, where `1` usually refers to the leftmost button on the mouse. For `<KeyPress>` and `<KeyRelease>` events the detail is a *keysym* (key symbol), which is a textual name describing a particular key on the keyboard. The keysym for an alphanumeric ASCII character such as `a` or `A` or `2` is just the character itself. Some other examples of keysyms are `Return` for the carriage-return key (often labeled "Enter" on modern keyboards), `BackSpace` for the Backspace key, `Delete` for the Delete key, and `Help` for the Help key. The keysym for a function key such as F1 is normally the same as the name that appears on the key. Section 22.6 contains a script that you can use to find out the keysyms for the keys on your keyboard.
Certain keys are defined as special *modifier keys*; these include the Shift keys, the Control key, the Command and Option keys on standard Mac keyboards, plus other keys such as Meta and Alt. The events just listed contain a *state* field that identifies which of the modifier keys were pressed at the time of the event and which mouse buttons were pressed. The state field is useful, for example, to trigger a script when the pointer moves with button 1 down, or when `Ctrl+A` is typed on the keyboard.
Events also contain other fields besides the ones described; the exact set of fields varies from event to event. See the reference documentation for a complete list of fields available for each event type.

22.3 An Overview of the `bind` Command

The `bind` command is used to create, modify, query, and remove bindings. This section illustrates the basic features of `bind`, and later sections go over the features in more detail.
Bindings are created with commands like this one:

```
bind .entry <Control-KeyPress-d> {.entry delete insert}
```

The first argument to the command specifies the path name of the window to which the binding applies. It can also be a widget class name such as `Entry`, in which case the binding applies to all widgets of that class (such bindings are called *class bindings*), or it can be `all`, in which case the binding applies to all widgets. You can also create your own symbolic binding tag, as described in Section 22.8. The second argument specifies a sequence of one or more events. The example specifies a single event, which is a keypress of the `d` character while the `Control` modifier key is down. The third argument may be any Tcl script. The script in the example invokes `.entry`'s widget command to delete the character just after the insertion cursor. The script is invoked if `Control+d` is typed when the application's input focus is in `.entry`. The binding can trigger any number of times. It remains in effect until `.entry` is deleted or the binding is explicitly removed by invoking `bind` with an empty script:

```
bind .entry <Control-KeyPress-d> {}
```

A *background error* occurs if there is an error during the execution of the script. The default behavior when a background error occurs in a Tk script is to post a dialog informing the user of the error. You can change this behavior, as described in Section 13.6.
The `bind` command can also be used to retrieve information about bindings. If `bind` is invoked with an event sequence but no script, it returns the script for the given event sequence:

```
    bind .entry <Control-KeyPress-d>
⇒ .entry delete insert
```

If `bind` is invoked with a single argument, it returns a list of all the bound event sequences for that window or class:

```
  bind .entry
⇒ <Control-Key-d>
  bind Button
⇒ <ButtonRelease-1> <Button-1> <Leave> <Enter> <Key-space>
```

The first example returned the bound sequences for `.entry`, and the second example returned information about all of the class bindings for button widgets. The class bindings were created by Tk's initialization script (the file `button.tcl` in the Tk library directory) to establish the default behavior for button widgets.

22.4 Event Patterns

Event sequences are constructed from basic units called *event patterns*, which Tk matches against the stream of events received by the application. An event sequence can contain any number of patterns, but most sequences have only a single pattern.
The most general form for a pattern consists of one or more fields between angle brackets, with the following syntax:

<modifier- modifier...- modifier- type-detail>

Whitespace may be used instead of dashes to separate the various fields, and most of the fields are optional. The *`type`* field identifies the particular event type, such as `KeyPress` or `Enter`. See Section 22.2 for a list of common event types. For example, the following script causes an entry widget `.entry` to change to a light green background whenever the mouse passes over it and to return to a white background when the button passes out of the widget again:

```
bind .entry <Enter> {.entry config -background lightgreen}
bind .entry <Leave> {.entry config -background white}
```

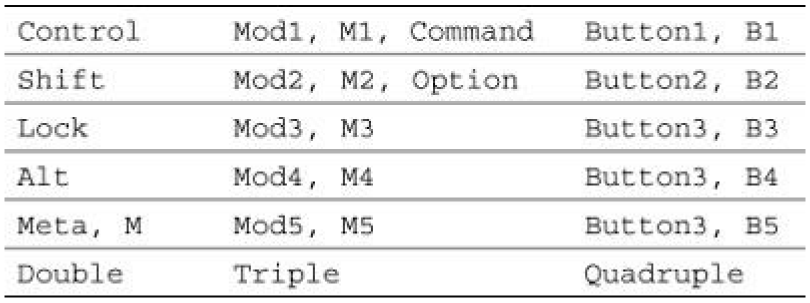
For key and button events the event type may be followed by a *detail* field that specifies a particular button or keysym. If no detail field is provided, as in `<KeyPress>`, the pattern matches any event of the given type. If a detail field is provided, as in `<KeyPress-Escape>` or `<ButtonPress-2>`, the pattern matches only events for the specific key or button. If a detail is specified, you can omit the `Key` or `Button` event types: `<Escape>` is equivalent to `<KeyPress-Escape>`.

Note

The pattern `<1>` is equivalent to `<Button-1>`, not `<KeyPress-1>`.

The event type may be preceded by any number of modifiers, each of which must be one of the values given in Table 22.1. If modifiers are specified, the pattern matches only events that occur when the specified modifiers are present. For example, the pattern `<Shift-Control-d>` requires that both the Shift and Control keys be held down when `d` is typed, and `<B1-Motion>` requires that the pointer move when mouse button 1 is down.

Table 22.1 Modifier Names for Event Patterns; Comma-Separated Modifiers Are Equivalent



Control	Mod1, M1, Command	Button1, B1
Shift	Mod2, M2, Option	Button2, B2
Lock	Mod3, M3	Button3, B3
Alt	Mod4, M4	Button3, B4
Meta, M	Mod5, M5	Button3, B5
Double	Triple	Quadruple

Not all modifiers are available on all systems. For example, the `Command` and `Option` modifiers are equivalent to `Mod1` and `Mod2` respectively and correspond to Macintosh-specific modifier keys. Some non-Macintosh systems might have keyboards with Meta keys that map to these modifiers, but many systems may not have keys that generate the `Mod1`-`Mod5` modifiers.
The "modifiers" `Double`, `Triple`, and `Quadruple` are used for specifying double, triple, and quadruple mouse clicks and other repeated events. They match a sequence of two, three, or four events, each of which matches the remainder of the pattern. For example, `<Double-1>` matches a double-click of mouse button 1, and `<Triple-2>` matches any triple-click of button 2.
A special shortcut form is available for patterns that specify key presses of printing ASCII characters such as `a` or `@`. You can specify a pattern for these events using just the single character. For example,

```
bind .entry a {.entry insert insert a}
```

arranges for the character `a` to be inserted into `.entry` at the point of the insertion cursor if it is typed when `.entry` has the keyboard focus. This command is identical to the command

```
bind .entry <KeyPress-a> {.entry insert insert a}
```

22.5 Sequences of Events

An event sequence consists of one or more event patterns optionally separated by whitespace. For example, the sequence `<Escape>a` contains two patterns. It triggers when the `Escape` key is pressed and then the `a` key is pressed.
A sequence need not necessarily match consecutive events. For example, the sequence `<Escape>a` matches an event sequence consisting of a `KeyPress` on `Escape`, a `KeyRelease` of `Escape`, and then a press of `a`; the release of `Escape` is ignored in determining the match. Tk generally ignores conflicting events in the input event stream unless they have the same type as the desired event. Thus, if some other key is pressed between the `Escape` and the `a`, the sequence does not match. As another example, the sequence `aB` matches a press of the `a` key, a release of the `a` key, a press of the `Shift` key, and a press of the `b` key. In this case, the `Shift` key press is ignored because it is a modifier key. Also, if several `Motion` events occur in a row, only the last one is used for matching binding sequences.

22.6 Substitutions in Scripts

The script that handles an event often needs to use some of the fields in the event, such as the pointer coordinates or the particular keysym that was pressed. Tk provides this access by substituting fields from the event wherever there are `%` characters in the script, in a fashion much like the Tcl `format` command. Before evaluating the script for a binding, Tk generates a new script from the original one by replacing each `%` and the character following it with information about the event. The character following the `%` selects a specific substitution to make. There are about 30 substitutions defined, but Table 22.2 lists the most commonly used ones. Not all substitutions are available for each event type. See the reference documentation for complete details of the substitutions available for different event types and their meanings.

Table 22.2 Common Event Detail Substitutions

Sequence	Substitution
%%	Replaced with a single percent
%b	The number of the button that was pressed or released. Valid only for ButtonPress and ButtonRelease events.
%d	The event detail in an event-specific format for built-in events. For virtual events generated with the event generate command, this can be an arbitrary string value provided to the event generate command. See the reference documentation for more details.
%h	The window height value for the event
%t	The time value for the event
%w	The window width value for the event
%x	The x-coordinate of the mouse pointer
%y	The y-coordinate of the mouse pointer
%A	The Unicode character value corresponding to a KeyPress or KeyRelease event, or an empty string if the event is for a key such as Shift that doesn't have a Unicode equivalent.
%D	The delta value of a MouseWheel event, representing the rotation units the mouse wheel has been moved. The sign of the value represents the direction the mouse wheel was scrolled.
%K	The keysym from KeyPress or KeyRelease event
%W	The full widget path name of the widget receiving the event
%X	The x-coordinate of the mouse pointer relative to the root window
%Y	The y-coordinate of the mouse pointer relative to the root window

As an example of using substitutions, the following script implements a simple mouse tracker:

```
bind all <Enter> {puts "Entering %W at (%x,%y)"}
bind all <Leave> {puts "Leaving %W"}
bind all <Motion> {puts "Pointer at (%x,%y)"}
```

If you type these commands in `wish`, messages are printed as you move the pointer over the windows of the application. Or you can use the following script to determine the keysyms for the keys on your keyboard:

```
bind . <KeyPress> {puts "The keysym is %K"}
focus .
```

If you type these two commands in `wish`, the name of the keysym will be printed for any key that you type. Try typing some normal keys like `a`, `A`, and `1`, plus special keys like F1, Return, or Shift.

Note

When Tk makes `%` substitutions, it treats the script as an ordinary string without any special properties. The normal quoting rules for Tcl commands are not considered, so `%` sequences are substituted even if embedded in braces or preceded by backslashes. The only way to prevent a `%` substitution is to double the `%` character.

22.7 Conflict Resolution

It is possible for several bindings to match a single event. For example, suppose there are bindings for `<Button-1>` and `<Double-Button-1>`, and button 1 is clicked three times. The first button press event matches only the `<Button-1>` binding, but the second and third presses match both bindings. The way conflicts are handled depends on how the bindings were declared. If the same tag is used for all the matching bindings, as in

```
bind .b <Button-1> ...
bind .b <Double-Button-1> ...
```

exactly one binding triggers, and it is the most specific of all the matching bindings. `<Double-Button-1>` is more specific than `<Button-1>`, so its script is executed on the second and third presses. Similarly, `<Escape>a` is more specific than `<Key-a>`, `<Control-Key-d>` is more specific than `<Key-d>`, and `<Key-d>` is more specific than `<KeyPress>`.
If more than one binding matches a particular event and the bindings have the same tag, the most specific binding is chosen and its script is evaluated. The following tests are applied, in order, to determine which of several matching sequences is more specific:

1. An event pattern that specifies a specific button or key is more specific than one that doesn't.
2. A longer sequence (in terms of number of events matched) is more specific than a shorter sequence.
3. If the modifiers specified in one pattern are a subset of the modifiers in another pattern, the pattern with more modifiers is more specific.
4. If these tests fail to determine a winner, the most recently registered sequence is the winner.

As an example, consider a label widget given only a single binding for the event `<Button-1>`:

```
label .l -text "Sample 1"
bind .l <Button-1> {
    puts "Left mouse button pressed"
}
```

This event matches a `<Button-1>` event and a `<Shift-Button-1>` event. In the following case, the label is given two bindings, one with the `Shift` modifier and one without:

```
label .l -text "Sample 2"
bind .l <Button-1> {
    puts "Left mouse button pressed"
}
bind .l <Shift-Button-1> {
    puts "Shift-Left mouse button pressed"
}
```

Because this widget has two bindings, event matching distinguishes the two different cases, and the presence of the more specific binding prevents the binding without the modifier from triggering if it matches.

22.8 Event-Binding Hierarchy

So far we've focused on bindings for a specific window, but actually there is a hierarchy of bindings that process events. Each window has a list of *bindtags*, also known simply as *tags*. Each bindtag in the list can contain a set of event bindings. Suppose that there are matching bindings with different tags, as in the following example:

```
bind all <Return> ...
bind .b <Any-KeyPress> ...
```

If the Return key is pressed, both of these bindings match. The different tags are handled independently, so that the matching binding triggers for each kind of tag. The order in which these tags are evaluated is defined with the `bindtags` command:

bindtags *widget* ?*tagList*?

The *`tagList`* is a list of tags in the order in which they are searched for event matching. By default, a widget is given four tags in the following order: the widget itself, the widget's class name, the toplevel window containing the widget, and `all`. If the `bindtags` command is issued with a single argument, it returns the current tag list for the specified widget. So for the widget `.b` in the preceding example, the default tag list is

```
bindtags .b
⇒ .b Button . all
```

This means that the binding for `.b` triggers first, followed by the binding for `all` when the Return key is pressed.
The tag list can be modified to change the tag order, or to add or remove windows, classes, or even user-defined tags. For example, since the behavior of a button is defined with bindings on the `Button` class, new behavior of a button can be assigned to a new tag, say, `SpecialButton`; simply changing the bindtags for a given button instantly changes the button's behavior, like so:

```
bind SpecialButton <Return> { # Some script... }
bindtags .b {.b SpecialButton . all}
```

Sometimes it's necessary to prevent event bindings farther up the hierarchy from triggering. This can be done by using the `break` command or the `return -code break` command. Whenever the binding code for an event returns a "break" status, further processing of the bindtags list is halted. For example, changing the previous example like this:

```
bind all <Return> {puts "all return!"}
bind .b <Return> {puts ".b return!"; break}
```

changes the behavior so that the `all` binding is not triggered, at least not for the `.b` window. The `break` command can be used within the body of the bind script. If the bind script calls a Tcl procedure, the procedure must use the `return -code break` command to return a break condition from the procedure itself to have the same effect.

22.9 When Are Events Processed?

Tk processes events only at a few well-defined times. After a Tk application completes its initialization, it enters an *event loop* to wait for window system events and other events such as timer and file events. When an event occurs, the event loop executes C or Tcl code to respond to that event. Once the response has completed, control returns to the event loop to wait for the next interesting event. Almost all events are processed from the top-level event loop. New events are not considered while the event loop responds to the current event, so there normally is no danger of one binding triggering in the middle of the script for another binding. This approach applies to all event handlers, including those for bindings, those for the script options associated with widgets, and others yet to be discussed, such as window manager protocol handlers.

Note

A few special commands, `tkwait`, `vwait`, and `update`, explicitly invoke the event loop. If called from within an executing event handler, they invoke nested instances of the event loop, which often causes unexpected effects. The best practice is not to call these commands from within your event-handling scripts. Commands and procedures that invoke the event loop should be specially noted in their reference documentation, so watch out for this. All other commands complete immediately without reentering the event loop.

Event handlers are always invoked in the global scope of the global namespace, even if the event loop was invoked from a `tkwait`, `vwait`, or `update` command inside a procedure. This means that you should use only global variables or fully qualified namespace variables in binding scripts. The context in which a handler is evaluated is not the same as the context in which the handler was defined. For example, suppose that you invoke `bind` from within a procedure. When the script in the binding eventually triggers, it does not have access to the local variables of the procedure; in most cases the procedure will have returned before the binding is invoked, so the local variables don't even exist anymore.

The best practice is to use Tcl procedures to implement a binding script, especially if the script needs to use temporary variables in order to process the event. This keeps the global namespace from becoming too cluttered with temporary variables. It can also prevent latent bugs when temporary variables are inadvertently shared. Another good practice for sharing variables is to use namespaces. (Namespaces are discussed in depth in Chapter 10.) Namespace variables can be shared by namespace-defined procedures, thereby isolating potential inadvertent interaction problems.

There is no problem having a bind script call a namespace-defined procedure; simply use a fully qualified name when calling the procedure, as shown here:

```
namespace eval SpecialButton {
    variable last_x
    variable last_y
    proc b1_event {W x y} {
        variable last_x
        variable last_y
        set last_x $x
        set last_y $y
        puts "Button-1 pressed"
        puts "at location $x,$y"
        puts "in window $W"
}

bind .b <Button-1> {SpecialButton::b1_event %W %x %y}
```

The other advantage to using Tcl procedures is performance. The Tcl interpreter bytecode compiles procedure bodies for performance. Because of the substitutions that take place each time a binding is triggered, a binding script cannot be bytecode-compiled. When a Tcl procedure is invoked from a binding, the procedure body can be compiled, improving the performance of the event handling.

22.10 Named Virtual Events

Tk's bindings provide an effective way to link actions from the windowing system or from user input to tasks within the application. This is possible with *named virtual events*. Let's look at how these may be put to use. Here's an example of a typical application binding:

```
bind .text <Control-c> {tk_textCopy %W}
```

Now this binding may be repeated for other widgets in the application as well. Windows applications commonly use `<Control-c>` for a copy shortcut; however, on the Apple Macintosh, this operation is commonly performed using `<Command-c>`. It would be cumbersome and error-prone to have to modify every event binding in order to simply change the shortcut key for a given operation. By using virtual events, we can simplify keyboard shortcut assignments. First, create the binding to a named virtual event:

```
bind .text <<Copy>> {tk_textCopy %W}
```

You would do this for every widget or widget class that supports the copy action. Next, tie the physical key binding to the named virtual event using the Tk `event` command:

```
event add <<Copy>> <Control-c>
```

This needs to be done only once. Defining a new shortcut is as easy as adding a new event definition:

```
event add <<Copy>> <Command-c>
```

So now there are two shortcuts for the copy operation, `<Control-c>` and `<Command-c>`. The former shortcut can be removed with the `event delete` command:

```
event delete <<Copy>> <Control-c>
```

Of course, there are situations where having both platforms' event sequences associated with the virtual event could cause incompatibilities. In cases like these, you can use the `tk windowingsystem` command, described in Chapter 29, to determine on which windowing system your application is running, and then set up only the virtual event definitions appropriate for that platform. For example, consider the case of binding scripts to a "right mouse click." Most Windows and Unix systems use three-button mice, and the right-click is delivered as a `<ButtonPress-3>` event. However, it is common for mice on Macintosh systems to have only a single button, and holding down the Control key while clicking is often used as the equivalent of a right-click. And even on those Macintosh systems that have a multibutton mouse, the right-click is delivered as a `<ButtonPress-2>` event. So you could define a virtual event named `<<RightClick>>` as follows:

```
if {[tk windowingsystem] eq "aqua"} {
    # Set up Mac OS X virtual event definitions
    event add <<RightClick>> <Button-2> <Control-Button-1>
} else {
    # Set up Windows and Unix virtual event definitions
    event add <<RightClick>> <Button-3>
}
```

If a virtual binding has the same sequence as a physical binding, the physical binding always takes precedence. So if both

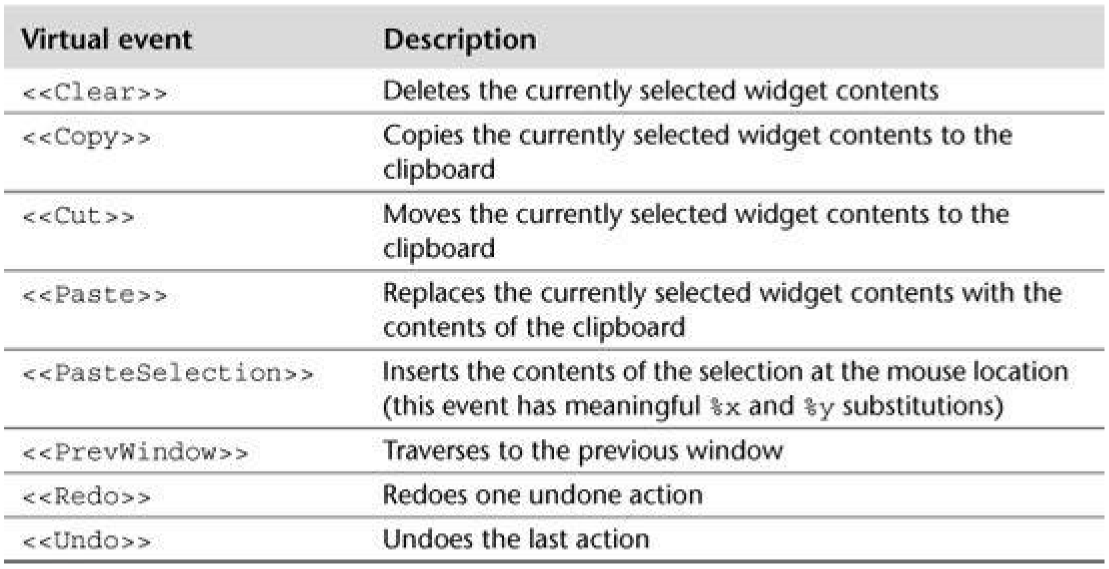
```
bind .text <Control-C> {puts "copy %W"}
```

and

```
bind .text <<Copy>> {tk_textCopy %W}
event add <<Copy>> <Control-c>
```

exist, the `puts` command executes, as this physical binding takes precedence over the `<<Copy>>` virtual binding.
Tk predefines some common operations as virtual events for widget bindings. The defaults for these are already platform-appropriate, so it is unlikely that your application needs to change them. Table 22.3 lists the predefined virtual events in Tk for user actions.

Table 22.3 Predefined Tk Virtual Events for User Actions



Virtual event	Description
`<<Clear>>`	Deletes the currently selected widget contents
`<<Copy>>`	Copies the currently selected widget contents to the clipboard
`<<Cut>>`	Moves the currently selected widget contents to the clipboard
`<<Paste>>`	Replaces the currently selected widget contents with the contents of the clipboard
`<<PasteSelection>>`	Inserts the contents of the selection at the mouse location (this event has meaningful `%x` and `%y` substitutions)
`<<PrevWindow>>`	Traverses to the previous window
`<<Redo>>`	Redoes one undone action
`<<Undo>>`	Undoes the last action

22.11 Generating Events

In addition to managing named virtual events as described earlier, the `event` command has the `generate` option, which can generate any windowing system event and send it to any window in the application, as if it came from the windowing system. Being able to generate events provides a means of testing an application's user interface. The syntax of `event generate` is

event generate *widget event* ?*option value option value* ...?

Button presses, mouse motions, and key presses can all be emulated with the `event generate` command. The *`option value`* pairs define the attributes of the event, such as the mouse position. (See the reference documentation for a complete list of supported options.) For example, the following command generates a mouse button event:

```
event generate .copy <ButtonPress> -button 1 -x 10 -y 10
```

In this case, the press is for button 1, normally the leftmost mouse button. In addition, the command provides the mouse location relative to the upper-left corner of the widget, which Tk uses for event-handling scripts with `%x` and `%y`. In this example, the event is delivered to the widget `.copy`, but the mouse pointer remains in its position on the screen. The `event generate` command also supports a `-warp` option, which, if provided a Boolean true value, also moves the mouse pointer.
You can also generate virtual events. The virtual events you generate don't even have to have a physical event sequence associated with them. This can serve as the basis of an asynchronous message-passing system in your application. For example, consider the following code:

```
label .display -textvariable msg -width 30 -anchor w
grid .display -sticky ew
bind .display <<Message>> {
    set msg "[clock format %t -format %%T]: %d"
}
event generate .text <<Message>> \
    -data "Hello" -time [clock seconds]
```

In this example, the widget named `.display` receives a `<<Message>>` event. The `-data` option allows you to pass an arbitrary string value with the event, which the event binding can retrieve via the `%d` event substitution; the `-time` option allows you to set a timestamp that can be retrieved with the `%t` substitution. In this example, the event handler simply formats the information received and sets the value of the label's linked text variable.
Another use of the `event generate` command is to tie a command action to a widget's virtual binding; for example:

```
    button .copy -text Copy -command {
        event generate [focus] <<Copy>>
    }
```

or

```
    .menu.edit add command -text Copy \
        -command {event generate [focus] <<Copy>>}
```

or

```
.menu.edit add command -text Copy \
        -command {event generate [focus] <<Copy>>}
```

In this way, a single button or menu command can be tied to whichever widget has the keyboard focus at the time the button is pressed.

22.12 Logical Actions

Most of the bindings discussed so far deal with interactions between the application and the user or between the application and the windowing system. Named virtual events and bindings can also be employed as a means of messaging or signaling between various widgets and the application. This is best illustrated through an example. We'll use a simple text editor to demonstrate how virtual events can be employed between the text widget and an application toolbar.
The text editor example contains a toolbar with buttons for copy, cut, and paste operations. When a selection is made in a text widget, the cut and copy buttons in the application's toolbar, as well as the Cut and Copy menu items in the Edit menu, should change to an enabled state. The text widget automatically generates a `<<Selection>>` event whenever its selection changes. This allows each of the widgets that care about selection to create a binding on `<<Selection>>` and update their states accordingly:

```
bind all <<Selection>> {updateToolbar %W}
```

In this example, the code binds to all widgets, so that no matter where the selection may occur, the toolbar gets properly updated. In the code that follows, the toolbar buttons are enabled and disabled based on whether text is selected.
The `updateToolbar` procedure examines the selection state of the widget `%W` to see if the selection is empty or not. It uses two other virtual events,

<<MayCopy>> and <<MayCut>>, to propagate the selection state to all the widgets that need to know:

```
proc updateToolbar {w} {
    set class [winfo class $w]
    if {$class eq "Text"} {
        set select_range [$w tag ranges sel]
        if {[llength $select_range] > 0} {
            set state normal
        } else {
            set state disabled
        }
        event generate $w <<MayCopy>> -data $state
        event generate $w <<MayCut>> -data $state
    }
}
```

At this point, you could simply hard-code the binding scripts for the <<MayCopy>> and <<MayCut>> events to update the appropriate menu entries and toolbar buttons. However, you might envision a modular application that could be extended with optional add-ons. These add-ons might need notification about when copies and cuts may take place. In this case, taking advantage of the bind command's ability to extend an existing binding script is quite useful. Remember that if the script argument to bind has + as its first character, the script is appended to any existing binding script rather than overwriting it. Thus, we could have our components register their interest in the copy/cut notification like this:

```
bind all <<MayCopy>>  \
    "+.toolbar.copy configure -state %d"
bind all <<MayCut>>   \
    "+.toolbar.cut configure -state %d"
bind all <<MayCopy>> \
    "+.mbar.edit entryconfigure 0 -state %d"
bind all <<MayCut>>  \
    "+.mbar.edit entryconfigure 1 -state %d"
```

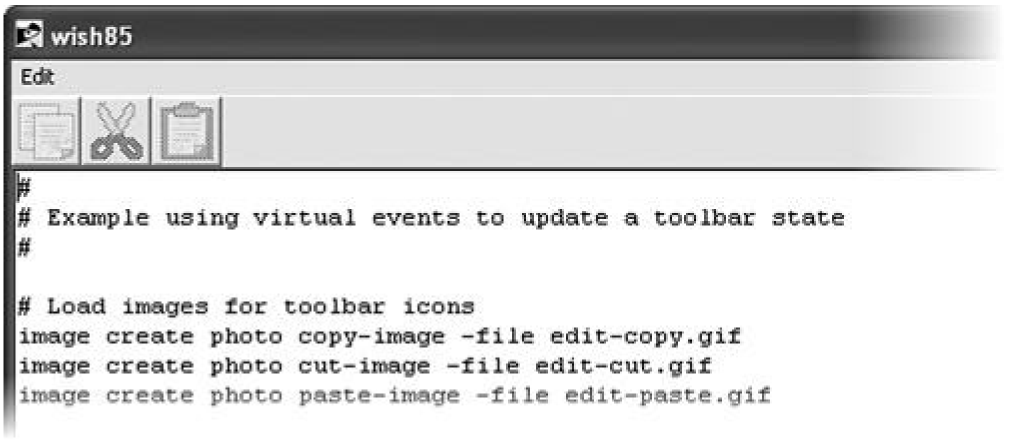
Now when the <<MayCopy>> or <<MayCut>> event is generated, the code updates toolbar buttons to copy and cut text, respectively. The code also updates the menu items to copy and cut text.
Likewise, binding the text widget to a <<Cut>> event and having the toolbar button and menu entry command generate a <<Cut>> event associate the action with the widget without one window having to have direct knowledge of the other. Note that the text widget already contains a binding for the <<Copy>> and <<Cut>> virtual events, so all that is needed is to generate this event when the

toolbar button is pressed:

```
button .toolbar.copy \
    -image copy-image \
    -command {event generate [focus] <<Copy>>}
button .toolbar.cut \
    -image cut-image \
    -command {event generate [focus] <<Cut>>}

.mbar.edit add command -label Copy
    -command {event generate [focus] <<Copy>>}
.mbar.edit add command -label Cut \
    -command {event generate [focus] <<Cut>>}
```

With this technique, the code merely issues an event indicating a cut or copy operation, letting the Tk text widget handle the rest. The `focus` command tells the code which widget has the text focus. That widget is the one used for copying or cutting text. Like this example, Tk's virtual events provide a surprising amount of power, especially when you make use of the preexisting body of Tcl code designed to handle these events.
Figure 22.1 shows an example of the editor in its initial state, with no text selected. Note how the toolbar buttons are disabled. As soon as something is selected, the copy and cut toolbar buttons become enabled, as shown in Figure 22.2.

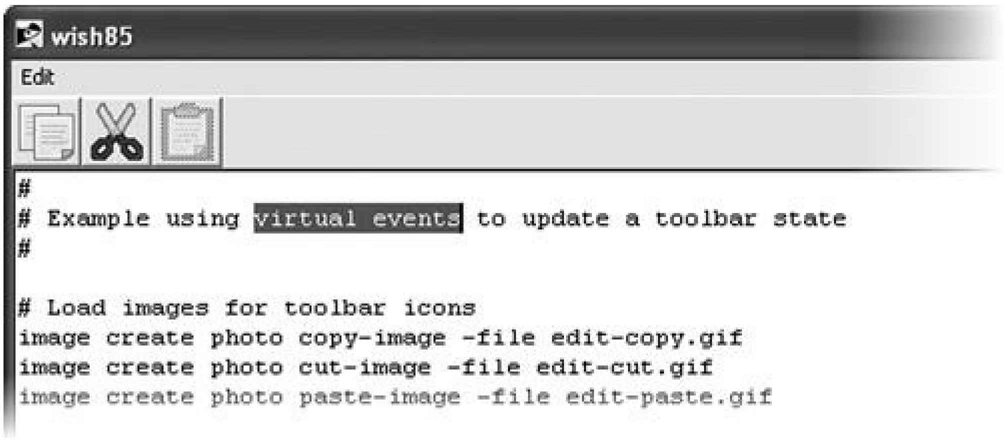
Figure 22.1 The toolbar buttons before selecting text. The icons in this figure are from the Tango Desktop Project (http://tango.freedesktop.org/Tango_Desktop_Project).



```
#
# Example using virtual events to update a toolbar state
#

# Load images for toolbar icons
image create photo copy-image -file edit-copy.gif
image create photo cut-image -file edit-cut.gif
image create photo paste-image -file edit-paste.gif
```

Figure 22.2 The toolbar buttons after text is selected. The icons in this figure are from the Tango Desktop Project

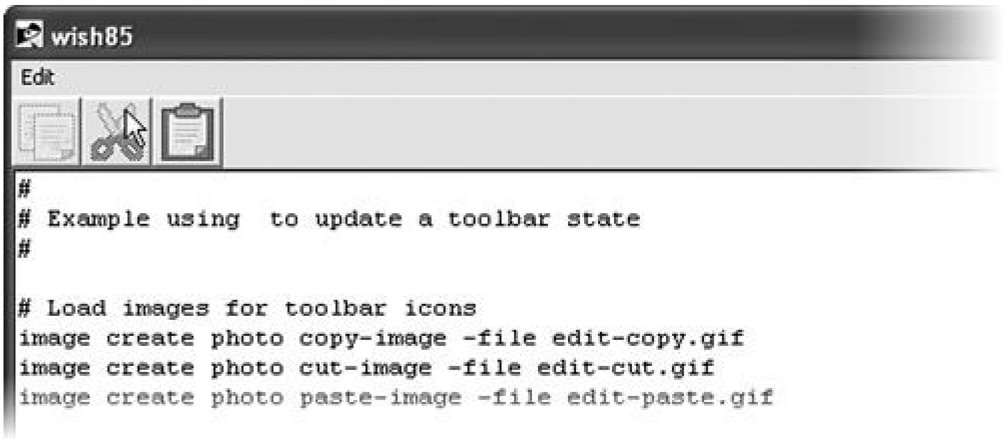
(http://tango.freedesktop.org/Tango_Desktop_Project).



```
#
# Example using virtual events to update a toolbar state
#

# Load images for toolbar icons
image create photo copy-image -file edit-copy.gif
image create photo cut-image -file edit-cut.gif
image create photo paste-image -file edit-paste.gif
```

After some text is selected, the cut button is pressed. At this point the copy and cut buttons become disabled because there is nothing selected, and the paste button becomes enabled because there is now text in the clipboard. Figure 22.3 reflects this state.

Figure 22.3 The toolbar buttons after the selected text is cut. The icons in this figure are from the Tango Desktop Project (http://tango.freedesktop.org/Tango_Desktop_Project).



```
#
# Example using  to update a toolbar state
#

# Load images for toolbar icons
image create photo copy-image -file edit-copy.gif
image create photo cut-image -file edit-cut.gif
image create photo paste-image -file edit-paste.gif
```

As applications grow more complex, the number of widget interactions can make user interface development a daunting task. When widget commands and states are organized into logical actions, the interactions become manageable.

22.13 Other Uses of Bindings

The binding mechanism described in this chapter applies to widgets. However, similar mechanisms are available internally within some widgets. For example, canvas widgets allow bindings to be associated with graphical items such as rectangles or polygons, and text widgets allow bindings to be associated with ranges of characters. These bindings are created using the same syntax for event sequences and % substitutions, but they are created with the widget command for the widget and refer to the widget's internal objects instead of to windows. See Chapter 23 for more information on the canvas widget and Chapter 24 for more information on the text widget.

23. The Canvas Widget

This chapter describes Tk's canvas widget. Canvases allow you to display and manipulate a variety of graphical objects such as rectangles, lines, images, and text strings. The canvas widget supports powerful features of allowing you to *tag* objects and manipulate all the objects with a given tag. For example, you can move or recolor all of the items with a given tag. You can also create event bindings for tags so that a Tcl script is invoked whenever the pointer passes over a tagged rectangle in a canvas, whenever a button is pressed over a string in the widget, and so on. The tagging and binding mechanisms make it easy to "activate" text and graphics so that they respond to the mouse. This chapter provides an introduction to canvases and shows some examples. It doesn't cover every detail, however; for that you should consult the reference documentation.

23.1 Canvas Basics: Items and Types

A canvas is a widget that displays a two-dimensional drawing surface on which you can place *items* of various *types*. Tk currently supports the following types:

- `rectangle`—drawn as an outline or a filled area; you can specify the outline color, outline width, fill color, and fill stipple pattern.
- `oval`—for circles and ellipses, with the same attributes as rectangles.
- `line`—consisting of one or more connected segments, a width, a color, a stipple pattern, and several other attributes such as whether to miter or round the corners between segments and whether or not to draw arrowheads at the ends of the line.
- Bézier curves—`line` items with an additional `-smooth` attribute specifying how Bézier cubic splines should be drawn instead of straight segments.
- `polygon`—consisting of three or more points, a fill color, and a stipple pattern. As with `line` items, you can also use the `-smooth` attribute to specify that the outline of the polygon should be a Bézier curve instead of a collection of straight segments.
- `arc`—drawn in any of several styles such as a pie wedge or a section of a circle. You can specify attributes such as the arc's angular

range, outline width, outline color, fill color, and fill stipple.

- `text`—consisting of one or more lines drawn in a single font. You can select a font, color, stipple pattern, justification mode, and line length.
- `bitmap`—consisting of a bitmap name, background color, and foreground color.
- `image`—consisting of a photo or bitmap image created using the Tk `image create` command or any of the Tk image extensions.
- `window`—an embedded widget of any kind, in which case the canvas acts as a geometry manager for the widget. The canvas reference documentation and command syntax use the term *window* to refer to these embedded widgets.

To create a canvas widget, use the `canvas` command:

canvas *widgetName ?option value ...?*

The canvas widget supports most standard widget options, such as `-background` to set the background color and `-borderwidth` to set the width of the widget's border. You can also request the `-height` and `-width` for the canvas using any supported Tk screen distance, as described in Section 18.2.2, though the geometry manager used to display the canvas may override the requested size (as discussed in Chapter 21).

The `create` subcommand creates new items on the canvas. For example, here is a script that creates a canvas named `.c` and then creates a new rectangle item on it:

```
canvas .c
grid .c -sticky nsew
.c create rectangle 1c 2.5c 4c 4.75c -width 2m \
    -outline blue -fill yellow
```

The first argument after the `create` action gives the type of the item. Following the type are pairs of x- and y-coordinates. The coordinates are floating-point values specifying any screen distance supported by Tk, as described in Section 18.2.2. When the canvas widget is created, its upper left corner is the `0, 0` origin, and the coordinates increase going down and to the right. The x- and y-coordinates may be provided either as separate arguments or as a single list argument following the type.

For rectangles there must be exactly four coordinates giving the locations of opposing corners of the rectangle. Ovals and arcs require four coordinates describing opposing corners of a rectangle bounding the entire oval. Lines accept coordinates for two or more points describing connected line segments. Polygons accept coordinates for three or more points describing the vertices; Tk automatically connects the last to the first point specified. All other types require coordinates of a single *positioning point* for the item. The `-anchor` option for those types then determines the location of the item positioned on the positioning point. For example, `-anchor nw` positions the upper left corner of the item on the positioning point.

Following the coordinates can be pairs of arguments specifying configuration options. The configuration options are specified in the same free-form style as for widgets. Default values apply for all options not specified. In the preceding example, the outline width, outline color, and fill color are specified, but no fill stipple is specified (it defaults to a solid fill). See the reference documentation for a complete list of all options supported for all types.

Items on the canvas have a *stacking order* (sometimes referred to as the *display list*). Items that appear lower in the stacking order (earlier in the list) might be obscured by items higher in the stacking order (later in the list). Creating a new item places it at the end of the list, on top of all other items in the canvas. As mentioned in the next section, you can also raise and lower items after creation, changing their stacking order.

Note

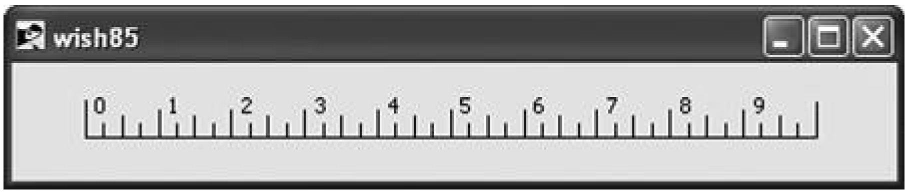
Window items (embedded widgets) are an exception to the stacking order rules. The underlying windowing systems require that they always be drawn on top of all other items; they are never obscured by other canvas items. If window items embedded within a canvas overlap each other, you can use the Tk `raise` and `lower` commands, discussed in Chapter 29, to change their stacking order relative to each other.

Here is a script that uses line and text items to create a simple ruler:

```
# ruler: draw a ruler on a canvas
canvas .c -width 12c -height 1.5c
pack .c
.c create line 1c 0.5c 1c 1c 11c 1c 11c 0.5c
for {set i 0} {$i < 10} {incr i} {
    set x [expr $i+1]
    .c create line ${x}c 1c ${x}c 0.6c
    .c create line $x.25c 1c $x.25c 0.8c
    .c create line $x.5c 1c $x.5c 0.7c
    .c create line $x.75c 1c $x.75c 0.8c
    .c create text $x.15c .75c -text $i -anchor sw
}
```

Figure 23.1 shows the canvas created by this script. For each centimeter of length, four tick marks are drawn with varying heights, along with a text item that displays the number of centimeters. No options are specified for the line items, since the default values are fine for this example. The option `-anchor sw` for the text items causes their lower left corners to be positioned at the specified coordinates.

Figure 23.1 The canvas generated by the `ruler` script



23.2 Manipulating Items with Identifiers and Tags

Each item on a particular canvas widget has a unique integer *identifier*. The identifier for an item is returned by the `create` subcommand, and it can be used in other widget commands to refer to the item. For example, you might invoke the command

```
set circle [.c create oval 1c 1c 2c 2c -fill black \
        -outline {}]
⇒ 12
```

to create a solid black circle and save its identifier (`12`, in this example) in the variable `circle`. Later on you could delete the item with the command

```
.c delete $circle
```

The widget command for a canvas provides several subcommands to manipulate items:

- You can `move` and `scale` items (but not rotate them, as of Tk 8.5).
- You can `raise` or `lower` them relative to one another (that is, change their z-order).
- You can query and change the coordinates (`coords`) and configuration options (`itemcget` and `itemconfigure`) for items.
- You can `find` the item nearest a given point, or all the items in a given rectangular region, or all the items with a given tag.
- For text items you can `insert` and delete (`dchars`) text, display and manipulate an insertion cursor (`icursor`), set the `focus`, and `select` a range of characters.
- You can `bind` events to them, such as `<ButtonPress>` or `<KeyPress>` events.

Consult the canvas reference documentation for a complete description of all subcommands and their use.
Canvases also provide a second way of referring to items, called *tags*. A tag is merely a text string associated with an item. A tag may have any form except that of an integer. A single item may have any number of tags, and a single tag may be applied to any number of items. Once an item has been tagged, you can use any of its tags to refer to it:

```
.c delete circle
```

In this example all of the items with the `circle` tag are deleted.
Tags serve three purposes in canvases. First, they allow you to use human-readable names for items so that you don't have to save item identifiers in variables. Second, they provide a grouping mechanism so that you can manipulate related items together. For example, consider the following commands:

```
.c itemconfigure circle -fill red
.c move circle 0 1c
```

The first command changes the fill color to red for all items with the `circle` tag, and the second command moves all the circle items down 1 centimeter. In general you can use a tag name anywhere that you can use an item identifier. You can also use simple logical expressions of tags, such as `circle&&element`; see the reference documentation for complete details.
The third use for tags is for defining behaviors. As discussed in the next section, bindings can be associated with tags. This allows groups of items

to be responsive to user events. You can also apply multiple tags to an item to combine several bound behaviors, or add and remove tags to change an item's behavior dynamically.
You can assign one or more tags to an item when you create it by providing a list of tags as the value of the `-tags` option, as in the following command:

```
.c create oval 1c 1c 2c 2c -fill black -outline {} \
        -tags {circle element}
```

This command assigns both `circle` and `element` as tags associated with the new item. You can also use the `itemconfigure` subcommand to assign a different list of tags to an existing item by changing the value associated with the `-tags` option.
The canvas widget command also provides commands for dynamically adding or removing tags on items. The `dtag` subcommand removes tags from use; for example:

```
# Remove the tag "circle" from item 12
.c dtag 12 circle

# Remove the tag "highlight" from all items with
# the tag "element"
.c dtag element highlight

# Remove the tag "element" from all items that use it
.c dtag element
```

You can also use the `addtag` subcommand to add a tag to one or more items:

canvas addtag *tagName searchSpec*

You need to specify the name of the tag to add as well as a search specification for identifying which items should be tagged. Table 23.1 shows the allowable formats for the search specification. You can also use the same format search specifications with the canvas `find` subcommand, which returns a list of identifiers for all items that match.

Table 23.1 Canvas Search Specifications

Specification	Function
`above` *tagOrID*	Selects the item just above the given item identified by another tag name or an ID
`all`	Selects all items on the canvas
`below` *tagOrID*	Selects the item just below the given item identified by another tag name or an ID
`closest` *x y ?offset? ?start?*	Selects the item closest to the given position. Any item closer than the *offset* to the position is considered to overlap the position. If multiple items are at the same distance or overlap at the point, the topmost one is selected. The *start* tag name or ID is used to find the closest item after the *start* item.
`enclosed` *x1 y1 x2 y2*	Selects all items enclosed within the given rectangle
`overlapping` *x1 y1 x2 y2*	Selects all items that overlap or are enclosed by the given rectangle
`withtag` *tagOrID*	Selects all items with the given tag or ID

The canvas widget automatically manages two additional predefined tags; you may not explicitly add or remove these tags. The tag `all` always refers to all items on the canvas. The tag `current` automatically refers to the topmost item whose drawn area covers the position of the mouse cursor. If the mouse is not in the canvas widget or is not over an item, no item has the `current` tag.

23.3 Bindings

When you create a canvas widget, you can create bindings in the normal way using the `bind` command; these bindings apply to the widget as a whole. (See Chapter 22 for a complete discussion of Tk events and using the `bind` command.) In addition, you can create bindings for individual items within a canvas using the `bind` subcommand. This command has the following form:

.c bind *itemOrTag sequence script*

The *`itemOrTag`* argument gives either the identifier of a single item, in which case the binding applies to that one item, or a tag name, in which case it applies to all items with the tag. *`sequence`* and *`script`* specify an event sequence and script just as for the `bind` command. The event sequence may use only `Enter` and `Leave` events (which trigger when the pointer moves into or out of an item), mouse motion events, button presses and releases, key presses and releases, or virtual events.

It is possible for multiple bindings to match a particular event, for example, if you've created bindings for multiple tags and then applied some combination of those tags to a particular item. When this occurs, all of the matching bindings are invoked, starting with any matching binding on the `all` tag, followed by any matching binding for each of the item's tags (in order), followed by any matching binding associated with the item's ID. If there are multiple matching bindings for a single tag, only the most specific binding is invoked. A `continue` command in a binding script terminates that script, and a `break` command terminates that script and skips any remaining scripts for the event, just as for the `bind` command.

Note

If bindings have been created for a canvas window using the `bind` command, they are invoked in addition to bindings created for the canvas's items using the `bind` subcommand. The bindings for items are invoked before any of the bindings for the widget as a whole.

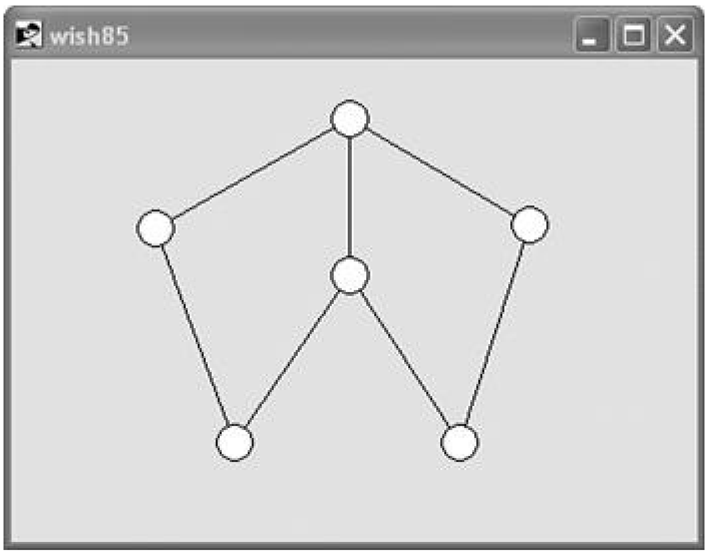
As an example of bindings, the following script allows you to create ball-and-stick graphs interactively on a canvas:

```
# graph: simple interactive graph editor

canvas .c
pack .c

# Creates a node (circle) at the given x, y position
proc mkNode {x y} {
    global nodeX nodeY edgeFirst edgeSecond
    set new [.c create oval [expr {$x-10}] [expr {$y-10}] \
            [expr {$x+10}] [expr {$y+10}] -outline black \
            -fill white -tags node]

    # Stores position of the node in two global arrays
    set nodeX($new) $x
    set nodeY($new) $y

    # Creates empty lists for edges in two global arrays
    set edgeFirst($new) {}
    set edgeSecond($new) {}
}

# Draw an edge (line) between two nodes (circles)
proc mkEdge {first second} {
    global nodeX nodeY edgeFirst edgeSecond
    set edge [.c create line $nodeX($first) $nodeY($first) \
            $nodeX($second) $nodeY($second)]
    .c lower $edge

    # Stores the edge ID in lists held in global arrays
    lappend edgeFirst($first) $edge
    lappend edgeSecond($second) $edge
}

# Draw a node at mouse position every time user clicks
# left mouse button
bind .c <ButtonPress-1> {mkNode %x %y}
```

```
# Highlight current item by changing its fill color
.c bind node <Enter> {
    .c itemconfigure current -fill black
}

# Remove highlight color
.c bind node <Leave> {
    .c itemconfigure current -fill white
}

# Typing "1" key stores item as first node for a line
bind .c <KeyPress-1> {
    set firstNode [.c find withtag current]
}

# Typing "2" key identifies end of the new line
bind .c <KeyPress-2> {
    set curNode [.c find withtag current]
    if {($firstNode != "") && ($curNode != "")} {
        mkEdge $firstNode $curNode
    }
}

# Assign keyboard focus to the canvas
focus .c
```

If you `source` this script into `wish`, you can create nodes by clicking on the canvas with button 1. You can create edges by moving the pointer over a node, typing `1`, then moving the pointer over a different node and typing `2`. See Figure 23.2 for an example of a graph created with this script.

Figure 23.2 A graph created interactively using the `graph` script



The script consists of two procedures for creating nodes and edges, plus five bindings. The `focus` command is also required to explicitly assign keyboard focus to the canvas widget. The script uses six global variables to hold information about the graph:

- `nodeX` and `nodeY` are associative arrays where the index is the item identifier for a node and the value is the x- or y-coordinate of the node's center. This information could be extracted from the canvas when needed, but it's simpler in this case to keep it in a Tcl array.
- `edgeFirst` and `edgeSecond` are associative arrays where the index is the identifier for a node and the value is a list of item identifiers for edges connecting to the node. `edgeFirst` holds the identifiers of all edges for which a node is the starting point, and `edgeSecond` identifies the edges for which a node is the ending point. This information is used below to drag nodes interactively.
- `firstNode` holds the identifier of the node that was under the pointer when `1` was typed last, or an empty string if `1` was typed when the pointer wasn't over a node.
- `curNode` holds the identifier of the node under the pointer or an empty string if the pointer isn't over a node.

The `mkNode` and `mkEdge` procedures just create new items in the canvas and record information about them in the global variables. The `lower` subcommand in `mkEdge` causes edges to be placed at the bottom of the display list for the canvas so that the nodes appear on top of them.

The first binding is set for the entire canvas using the `bind` command; it causes a new node to be created at the position of the pointer whenever button 1 is pressed. The `Enter` and `Leave` bindings apply to all the nodes in the canvas; they cause nodes to change color when the mouse passes over them. The last two bindings are used to create edges; they trigger whenever `1` or `2` is typed in the canvas. Both of these bindings make use of the special tag `current`, which is managed by Tk. The binding for `1` invokes the command

```
.c find withtag current
```

which returns a list of identifiers for the items that have the tag `current` (in this case the result is either a list with one element or an empty list) and saves the result in `firstNode`. The binding for `2` retrieves the current item and also creates a new edge.

The following script extends `graph` to allow nodes to be dragged interactively by holding the Control key while dragging with mouse button 1:

```
proc moveNode {node xDist yDist} {
    global nodeX nodeY edgeFirst edgeSecond
    .c move $node $xDist $yDist
    incr nodeX($node) $xDist
    incr nodeY($node) $yDist
    foreach edge $edgeFirst($node) {
        .c coords $edge $nodeX($node) $nodeY($node) \
                [lindex [.c coords $edge] 2] \
                [lindex [.c coords $edge] 3]
    }
    foreach edge $edgeSecond($node) {
        .c coords $edge [lindex [.c coords $edge] 0] \
                [lindex [.c coords $edge] 1] \
                $nodeX($node) $nodeY($node)
    }
}

.c bind node <Control-ButtonPress-1> {
    set curX %x
    set curY %y
}
.c bind node <Control-B1-Motion> {
    moveNode [.c find withtag current] [expr {%x-$curX}] \
            [expr {%y-$curY}]
    set curX %x
    set curY %y
}
# Do nothing on the canvas when receiving these events. This
# binding exists to prevent the "best match" algorithm from
# matching the <ButtonPress-1> canvas binding when all we
# want to trigger is the item binding.

bind .c <Control-ButtonPress-1> { }
```

The procedure `moveNode` does all the work of moving a node: it uses the `move` action to move the node item, then it updates all of the edges for which this node is an endpoint. For each edge `moveNode` uses the `coords` action to read out the edge's current coordinates and replace either the first two or last two coordinates to reflect the node's new location.

The two new bindings apply to all items with the tag `node`. When button 1 is pressed while the Control key is held, the pointer coordinates are stored in variables `curX` and `curY`. When the mouse is dragged with button 1 down while the Control key is held, the current item is moved by the amount the mouse has moved and new pointer coordinates are recorded.

Note that there is an additional widget binding for the `<Control-ButtonPress -1>` event as well, which provides a script that does nothing. If you omitted this binding, a `<Control-ButtonPress-1>` event would be presented to the canvas widget after the canvas item binding is processed. Without an exact binding match, Tk would look for the best match and find and execute the

`<ButtonPress-1>` binding on the canvas. As a result, every time you tried to move a node, you'd also end up creating a new node, which is not desired behavior.

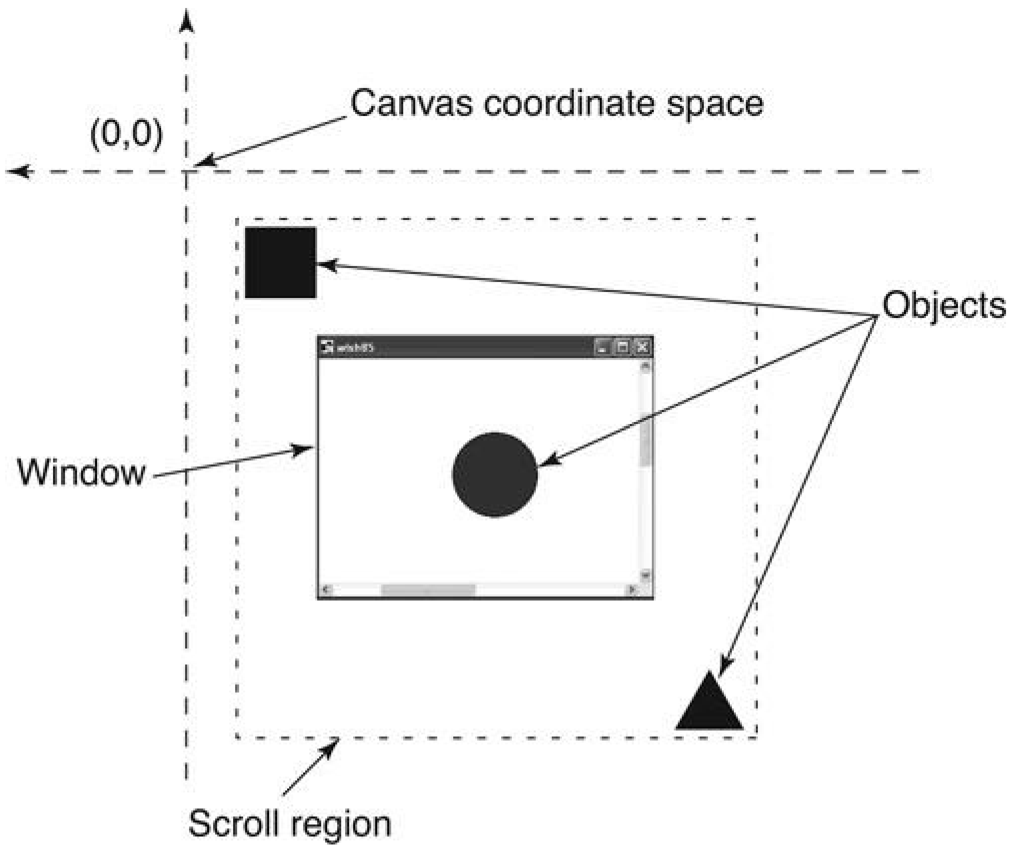
23.4 Canvas Scrolling

Canvases support vertical and horizontal scrolling in the standard fashion for Tk widgets (see Section 18.9). However, if a canvas is scrolled, the coordinates in the canvas window are not the same as the coordinates on the logical surface of the canvas. To handle this situation canvases provide additional widget command actions for translating from screen to canvas coordinates.

Since the canvas can be considered a virtual sheet at least 32,000 by 32,000 pixels in size, the scrolling is not enabled until a *scroll region* is defined. This limits the area that the scrollbars use as the visible region of the canvas. The scroll region is defined by setting the canvas `-scrollregion` option. The value is a list of four numbers defining the left, top, right, and bottom bounds of the region, respectively. The canvas widget uses these bounds to set the scrollbar range (see Figure 23.3). The most common way to set the scroll region is to use the canvas `bbox` subcommand to obtain the bounding region of all items currently on the canvas:

```
.c configure -scrollregion [.c bbox all]
```

Figure 23.3 Canvas scroll region



Note

Any time you create, delete, move, scale, or otherwise modify items on the canvas, you could end up modifying the bounding region of all items on the canvas.

Now that the canvas can be scrolled to view different regions of the virtual canvas, translating mouse x- and y-coordinates to canvas coordinates is necessary in order to identify locations in the canvas for the current view. Mouse x- and y-coordinates are measured relative to the window's upper left origin. If the canvas is scrolled, the upper left corner of the window no longer represents the `0, 0` coordinate on the canvas. The canvas widget's `canvasx` and `canvasy` subcommands translate the window-relative coordinates into the canvas coordinates based on the current scroll positions.
For example, to enable scrolling of the graph application presented in Section 23.3, you would need to modify all of the mouse event-binding scripts to translate from window-relative mouse coordinates to canvas coordinates, such as

```
bind .c <ButtonPress-1> {
    mkNode [.c canvasx %x] [.c canvasy %y]
}
```

23.5 PostScript Generation

Canvases can generate Encapsulated PostScript descriptions of their contents for printing and insertion into other documents. Use the `postscript` scommand to output the contents of a canvas widget to PostScript; for example:

```
.c postscript -file canvas.eps
```

This command writes the contents of the canvas widget `.c` into a file named `canvas.eps`. If you don't provide the `-file` option, the PostScript data becomes the return value of the command.
The `postscript` subcommand has a number of options to configure the PostScript results. See the canvas reference documentation for more information. The `pdf4tcl` extension, available at http://pdf4tcl.berlios.de/, allows you to output the contents of a canvas widget to a PDF file. Many users find the PDF format easier to work with than Encapsulated PostScript.

24. The Text Widget

This chapter describes Tk's text widget, which displays one or more lines of text. The text widget can also display embedded images and widgets. The text widget can be used to display text, provides the ability to edit text, and even presents an interface for interacting with HTML or other tagged text.

24.1 Text Widget Basics

You create a text widget with the `text` command:

text *widgetName ?option value ...?*

The text widget supports many common widget options, such as `-background`, `-foreground`, and `-font`, which determine the default settings of the text displayed; additional options specific to the text widget, such as `-tabs` and `-wrap`, provide default tab stops and line-wrapping settings. As you will see in Section 24.4, all of these settings can be overridden for particular characters by creating and applying *tags*. The text widget also supports `-height` and `-width` options for the height in lines of text and the width in characters; of course, the geometry manager used to display the canvas may override the requested size (as discussed in Chapter 21). The `-undo` option turns support for undo operations on or off, as described in Section 24.8. Text widgets also support vertical and horizontal scrolling in the standard fashion for Tk widgets (see Section 18.9). See the text widget reference documentation for a complete description of supported options.
The basic text widget subcommands for manipulating text content are as follows:

- `widget delete index1 ?index2 ...?`

Deletes a range of characters from the text, starting with the character at *`index1`* up to but not including the character at *`index2`*. If only *`index1`* is specified, that single character is deleted. Multiple ranges may also be specified.

- `widget get ?-displaychars? ?--? index1 ?index2 ...?`

Returns a range of characters from the text, up to but not including the character at *`index2`*. If only *`index1`* is specified, it returns that single character.

Multiple ranges may also be specified. Characters hidden with the `-elide` option are returned unless `-displaychars` is specified.

- `widget insert index chars ?tagList chars tagList ...?`

Inserts the string `chars` just before the character at `index`, optionally applying the tags listed in `tagList` to the characters. Multiple strings and tags may be provided for insertion.

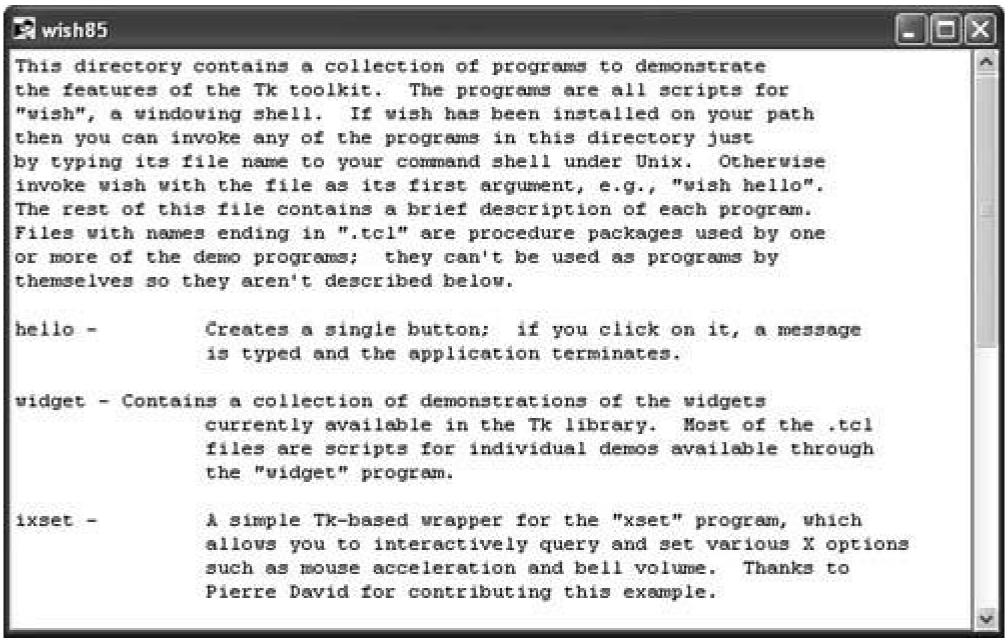
- `widget replace index1 index2 chars ?tagList chars tagList ...?`

Deletes a range of characters from the text, starting with the character at `index1` up to but not including the character at `index2`, replacing them with the string `chars` and optionally applying the tags listed in `tagList` to the characters. Multiple strings and tags may be provided for insertion.
Each line of text contained in a text widget must be terminated with a newline character. This is quite handy when reading text files and displaying their contents in a text widget. Depending on the `-wrap` settings that apply to the text, a single logical line might be wrapped visually in the text widget display, or it might be truncated and require horizontal scrolling to view the portions not visible.
As an example, the following script creates a text widget with an associated scrollbar and reads a file into the text widget:

```
# text: read a file into a text widget

text .text -relief raised -bd 2 \
        -yscrollcommand {.scroll set}
scrollbar .scroll -command {.text yview}
grid .text -row 0 -column 0 -sticky nsew
grid .scroll -row 0 -column 1 -sticky ns

proc loadFile {file} {
    .text delete 1.0 end
    set f [open $file]
    .text insert end [read $f]

    close $f
}
loadFile $tk_library/demos/README
```

The first group of lines in the script creates a text widget and a scrollbar, grids them side by side in the main window, and sets up connections between the text and the scrollbar. The next part of the script defines and calls the procedure `loadFile`, which opens a file, reads its content, and inserts the content at the end of the text widget. The `.text delete` command in `loadFile`, which deletes all of the text in the widget, isn't necessary in this example since the new widget is already empty. However, it allows `loadFile` to be invoked again later to replace the contents of the widget with a new file. After the script has completed, the screen should look like Figure 24.1 (assuming that your system has the standard Tk widget demos, which are part of a standard Tk distribution).

Figure 24.1 A text widget and associated scrollbar



```
This directory contains a collection of programs to demonstrate
the features of the Tk toolkit.  The programs are all scripts for
"wish", a windowing shell.  If wish has been installed on your path
then you can invoke any of the programs in this directory just
by typing its file name to your command shell under Unix.  Otherwise
invoke wish with the file as its first argument, e.g., "wish hello".
The rest of this file contains a brief description of each program.
Files with names ending in ".tcl" are procedure packages used by one
or more of the demo programs;  they can't be used as programs by
themselves so they aren't described below.

hello -         Creates a single button;  if you click on it, a message
                is typed and the application terminates.

widget - Contains a collection of demonstrations of the widgets
                currently available in the Tk library.  Most of the .tcl
                files are scripts for individual demos available through
                the "widget" program.

ixset -         A simple Tk-based wrapper for the "xset" program, which
                allows you to interactively query and set various X options
                such as mouse acceleration and bell volume.  Thanks to
                Pierre David for contributing this example.
```

The default bindings for text widgets allow you to manipulate the text in a number of ways:

- You can scroll the text with the scrollbar or by scanning with mouse button 2 in the text.
- You can edit the text; for example, click with mouse button 1 to set the insertion cursor, then type new characters.
- You can select information by dragging with mouse button 1 and then copying the selection into other applications.

See the reference documentation for text widgets for complete information on the default bindings.

24.2 Text Indices and Marks

Many of the text widget commands require you to identify particular places in the text. For example, the `insert` action in the script from Section 24.1

specified `end` as the place to insert the text read from the file. A position specifier in a text widget is called an *index*; it can take any of several forms. The simplest form for an index is two numbers separated by a dot, such as `2.3`. The first number gives a line number and the second number gives a character index within a line. The characters within a line are numbered starting with 0, as is common in Tcl; however, the lines are numbered starting with 1, so as to be compatible with most other programs that manipulate text files. If the character index is the string `end`, as in `5.end`, it refers to the newline character terminating the line. The index `end` refers to the end of the file, and an index in the form `@`*`x,y`*, where *`x`* and *`y`* are numbers, refers to the character closest to the pixel at location *`x,y`* in the window.
You can also use symbolic names for positions in a text; these symbolic names are called *marks*. For example, the command

```
.text mark set first 2.3
```

sets a mark named `first` to refer to the gap between character 3 of line 2 and the character just before it. In the future you can refer to the character after the mark as `first` instead of `2.3`. The mark continues to refer to the same logical position even if characters are added to or deleted from the text. For example, if you delete the first character of line 2, `first` becomes synonymous with index `2.2` instead of `2.3`.
Each mark also has a "gravity," which is either `left` or `right`. The gravity for a mark specifies what happens to the mark when text is inserted at the point of the mark. If a mark has left gravity, it is treated as if it were attached to the character on its left, so the mark remains to the left of any text inserted at the mark position. If the mark has right gravity, which is the default, new text inserted at the mark position appears to the left of the mark (so that the mark remains rightmost). You can query or set the gravity of a mark with the `mark gravity` subcommand, such as in this example, which sets the gravity of the mark `first` to `left`:

```
.text make gravity first left
```

Two marks have special meaning in text widgets. The `insert` mark identifies the location of the insertion cursor; if you modify `insert`, Tk displays the insertion cursor at the new location. The second special mark is `current`, which Tk updates continuously to identify the character underneath the mouse pointer.
Indices can also take more complex forms consisting of a base followed by one or more modifiers. For example, consider the following command:

```
.text delete insert "insert + 2 chars"
```

Both of the indices for this `delete` command use `insert` as a base, but the second index also has a modifier `+ 2 chars`, which advances the index by two characters over its base value. Thus, the command deletes the two characters just to the right of the insertion cursor. Some other examples of modifier usage are `first lineend`, which refers to the newline at the end of the line containing the mark `first`, and `insert wordstart`, which refers to the first character of the word containing the insertion cursor.

24.3 Search and Replace

The text widget supports a `search` subcommand and a `replace` subcommand. The `search` subcommand scans through the text looking for exact match strings or regular expression pattern matching. It has the ability to return the next match after a given starting point or returning all the matching locations. The `replace` subcommand combines delete and insert operations into a single command.
Here is a procedure that uses marks and other indices to provide a general-purpose searching facility for text widgets:

```
proc forAllMatches {w pattern script} {
    set countList [list]
    set startList [$w search -all -regexp \
            -count countList $pattern 1.0 end]
    foreach first $startList count $countList {
        $w mark set first $first
        $w mark set last [$w index "$first + $count chars"]
        uplevel 1 $script
    }
}
```

The `forAllMatches` procedure takes three arguments: the name of a text widget, a regular expression pattern, and a script. It uses the text widget’s `search` subcommand to find all the starting indices for the text matching the pattern. The `-count` option returns a list of the number of characters for each match in the variable `countList`. For each match, `forAllMatches` sets the marks `first` and `last` to the beginning and end of the range, then it invokes the script. For example, the following script prints out the locations of all instances of the word `Tk` in the text:

```
forAllMatches .text Tk {
    puts "[.text index first] --> [.text index last]"
}
```

```
⇒ 2.20 --> 2.22
  16.29 --> 16.31
  20.18 --> 20.20
  34.47 --> 34.49
```

For each matching range the `index` subcommand is invoked for the `first` and `last` marks; it returns numerical indices corresponding to the marks, which `puts` prints on standard output. As another example, you could clean up redundant `the` words in a text with the following script:

```
forAllMatches .text "the the" {
    .text delete first "first + 4 chars"
}
```

In this script the action taken for each range is to delete the first four characters of the range, which eliminates the redundant `the` (this example works only if both `the`s are on the same line).
Because the text widget indices look like real numbers, there is a tendency to treat them as real numbers, but this can cause unexpected problems. For example, comparing two indices using a simple expression does not give correct results:

```
.t mark x 3.37
set x [.t index x]
set mark_insert [.t index insert]
if {$mark_x > $mark_insert} {
    puts "Insert is before mark x"
} else {
    puts "Insert is after mark x"
}
```

In this example, if the insert mark was located at index 3.7, the code incorrectly reports that the insert mark was after the `x` mark.
The text widget provides a `compare` subcommand to perform index relational comparisons:

widget compare *index1 op index2*

This compares the indices given by *`index1`* and *`index2`* according to the relational operator given by *`op`* and returns `1` if the relationship is satisfied and `0` if it is not. *`op`* must be one of the operators `<`, `<=`, `==`, `>=`, `>`, or `!=`. Here is the correct way to compare the index values in the preceding example:

```
if {[.t compare x > insert]} {
    puts "Insert is before mark x"
} else {
    puts "Insert is after mark x"
}
```

24.4 Text Tags

Text tags provide a tagging mechanism similar to that of canvases except that text tags apply to ranges of characters instead of graphical items. Tags serve three purposes in the text widget. First, they are used to control the formatting of the characters, such as foreground and background color, font, spacing, and left and right margins. Second, they provide a way to manage the selection. Finally, they provide event bindings, turning plain text into active controls.
A tag name can have any string value, and it can be applied to arbitrary ranges within the text. A single character may have multiple tags, and a single tag may be associated with many ranges of characters. For example, the command

```
.text tag add theWholeEnchilada 1.0 1.end
```

applies the tag `theWholeEnchilada` to the first line of the text; the command

```
.text tag remove wrd "insert wordstart" "insert wordend"
```

removes the tag `wrd` from all the characters in the word around the insertion cursor; and

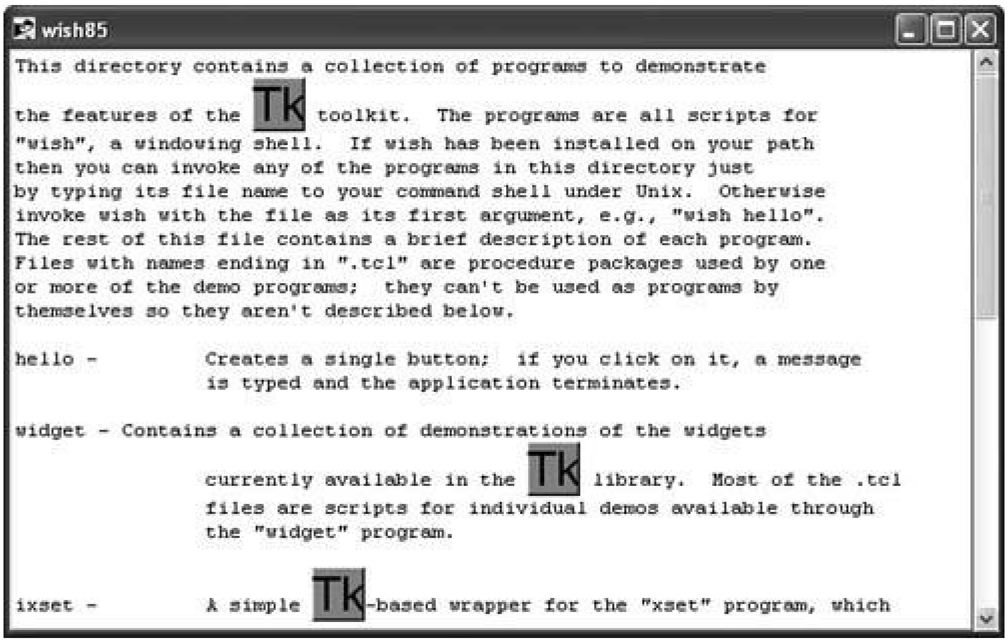
```
  .text tag ranges hot
⇒ 1.0 1.3 1.8 1.13
```

returns a list of indices for the beginning and end of each range of characters tagged with `hot`. In the preceding example the tag `hot` is present on the characters at indices 1.0 through 1.2 and 1.8 through 1.12.
Using the `forAllMatches` procedure from Section 24.3, let’s demonstrate how tags can be used to control the formatting of text. The following script changes all instances of the word `Tk` so they are drawn with a larger font, a darker background color, and a raised relief:

```
forAllMatches .text Tk {
    .text tag add big first last
}
.text tag configure big -background "cornflower blue" \
        -font {Helvetica 24} \
        -borderwidth 2
        -relief raised
```

The results are shown in Figure 24.2.

Figure 24.2 Using tags to format characters in a text widget



```
This directory contains a collection of programs to demonstrate

the features of the Tk toolkit.  The programs are all scripts for
"wish", a windowing shell.  If wish has been installed on your path
then you can invoke any of the programs in this directory just
by typing its file name to your command shell under Unix.  Otherwise
invoke wish with the file as its first argument, e.g., "wish hello".
The rest of this file contains a brief description of each program.
Files with names ending in ".tcl" are procedure packages used by one
or more of the demo programs;  they can't be used as programs by
themselves so they aren't described below.

hello -         Creates a single button;  if you click on it, a message
                is typed and the application terminates.

widget - Contains a collection of demonstrations of the widgets

                currently available in the Tk library.  Most of the .tcl
                files are scripts for individual demos available through
                the "widget" program.


ixset -         A simple Tk-based wrapper for the "xset" program, which
```

One special tag, sel, is used to manage the selection and has these special characteristics:

- Whenever characters are tagged with sel, the text widget claims ownership of the selection.
- Attempts to retrieve the selection are serviced by the text widget, returning all the characters with the sel tag.
- If the selection is claimed away by another application or by another window within the current application, the sel tag is removed from all characters in the text.
- Whenever the sel tag range changes, a virtual event <<Selection>> is generated. This is discussed more in Section 24.5.
- The sel tag cannot be deleted with the tag delete command.
- The sel tag is not shared with peer text widgets. Instead, each text widget maintains its own sel tag. Peer text widgets are discussed in Section 24.9.

24.4.1 Tag Options

There are almost two dozen tag options that can be grouped into three categories: visibility, which allows for the hiding of text; formatting, which affects the look of the characters such as font, size, and color; and layout, which affects the placement of the characters on the display.
The only visibility tag option is `-elide`, which takes a Boolean value. If this value is true for a tag, the characters to which this tag is applied are not displayed by the text widget. By default, other text subcommands still apply to the elided characters, though many of them have an option to ignore elided characters if desired.
Formatting tag options are similar to named character styles in a word-processing application in that they affect the appearance of individual characters. Thus, a single line of text could have several tags applying formatting to groups of characters, achieving a mix of colors, fonts, sizes, and so on. The following formatting tag options are available:

- `-background`—the background color of the characters
- `-bgstipple`—a stipple pattern for the background
- `-borderwidth`—the width of the text border, as a screen distance
- `-fgstipple`—a stipple pattern applied to the characters
- `-font`—the character font
- `-foreground`—the color of the characters
- `-offset`—the number of pixels by which the text's baseline should be offset vertically from the overall line's baseline; a positive value raises the text to achieve a superscript effect, and a negative value lowers the text to achieve a subscript effect
- `-overstrike`—a Boolean specifying whether or not to draw an overstrike line through the middle of the characters
- `-relief`—the three-dimensional border style for the text
- `-underline`—a Boolean specifying whether or not to underline the characters

The layout tag options are somewhat analogous to named paragraph styles in a word-processing application, in that they determine display settings that affect entire display lines. The layout tag options differ from the formatting tag options in that they apply only if they occur on a tag applied to the first nonelided (visible) character on a display line. Because it is difficult to predict which characters might appear at the beginning of display lines, the common practice when using these tag options is to apply the relevant tag to the entire logical line of characters. The following layout tag options are available:

- -justify—how to justify display lines; one of left, right, or center
- -lmargin1—the indent from the left side of the widget for the first display line, expressed as a screen distance
- -lmargin2—the indent from the left side of the widget for the second and subsequent display lines, expressed as a screen distance
- -rmargin—the indent from the right side of the widget for display lines, expressed as a screen distance
- -spacing1—the amount of extra space to leave above the first display line, expressed as a screen distance
- -spacing2—the amount of extra space to leave above the second and subsequent display lines, expressed as a screen distance
- -spacing3—the amount of extra space to leave below the last display line, expressed as a screen distance
- -tabs—a list of tab stops, as discussed below
- -tabstyle—the tab style, as discussed below
- -wrap—whether to wrap at word boundaries (word), at any character (char), or not at all (none)

Tab stops for tags as well as the widget as a whole are set with the -tabs option, which accepts a list of tab stops expressed as screen distances from the left edge of the widget. After each position element in the list, you have the option of providing one of the following keywords as another element to set the justification of that tab stop: left (the default), right, center, or numeric. left causes the text following the tab character to be positioned with its left edge at the tab position, right means that the right edge of the text following the tab character is positioned at the tab position, and center means that the text is centered at the tab position. numeric means that the decimal point in the text is positioned at the tab position; if there is no decimal point, the least significant digit of the number is positioned just to the left of the tab position; if there is no number in the text, the text is right-justified at the tab position. For example,

```
-tabs {2c left 4c 6c center}
```

creates three tab stops at 2-centimeter intervals; the first two use left justification and the third uses center justification.

If the list of tab stops does not have enough elements to cover all of the tabs in a text line, Tk extrapolates new tab stops using the spacing and alignment from the last tab stop in the list. If no -tabs option is specified, or if it is specified as an empty list, Tk uses default tabs spaced every eight (average

size) characters.
The `-tabstyle` option specifies how to interpret the relationship between tab stops on a line and tabs in the text of that line. The value must be `tabular` (the default) or `wordprocessor`. If the tab style is `tabular`, the `n`th tab character in the line's text is associated with the `n`th tab stop defined for that line. If the tab character's x-coordinate falls to the right of the `n`th tab stop, the text widget displays a gap of a single space as a fallback. If the tab style is `wordprocessor`, any tab character being laid out uses the first tab stop to the right of the preceding characters already laid out on that line.

24.4.2 Tag Priorities

As mentioned previously, a range of characters can have multiple tags applied to it. If two or more tags specify options that conflict, the options of the tag with the highest *priority* are used. If a particular display option has not been specified for a particular tag, or if it is specified as an empty string, that option is not used; the next-highest-priority tag's option is used instead. If no tag specifies a particular display option, the default style for the widget is used.
When a new tag is defined, it is initially assigned a priority higher than that of any existing tags defined for the text widget. You can change the priority of a tag with the `tag lower` and `tag raise` subcommands. For example, the following command takes the `important` tag and raises it to the highest tag priority:

```
.text tag raise important
```

This next example lowers the priority of `bold` to just lower than the `URL` tag:

```
.text tag lower bold URL
```

24.4.3 Tag Bindings

The text widget allows you to associate bindings to tags with the `tag bind` subcommand. You can use bindings to make portions of text "active" so that they respond to mouse, keyboard, `<Enter>`, `<Leave>`, or virtual events. Among other things, this enables you to implement hypertext effects. For example, the following bindings cause all the `Tk` words to turn green whenever the mouse passes over any one of them:

```
.text tag bind big <Enter> {
    .text tag configure big -background SeaGreen2
}
.text tag bind big <Leave> {
    .text tag configure big -background Bisque3
}
```

The following binding causes the text widget to reload itself with the Tk demo's `README` file when the user holds the Control key while clicking button 1 on any of the `Tk` words:

```
.text tag bind big <Control-Button-1> {
    .text delete 1.0 end
    loadFile $tk_library/demos/README
}
```

Note

Keyboard events trigger a tag binding only when the mouse cursor is within the tagged region containing the binding. The insert cursor does not have to be within the tagged region.

It is possible for multiple bindings to match a particular event, for example, if you've created bindings for multiple tags and then applied some combination of those tags to a section of text. When this occurs, all of the matching bindings are invoked, in order from lowest-priority to highest-priority tag. If there are multiple matching bindings for a single tag, only the most specific binding is invoked. A `continue` command in a binding script terminates that script, and a `break` command terminates that script and skips any remaining scripts for the event, just as for the `bind` command.

Note

If bindings have been created for a text widget using the `bind` command, they are invoked in addition to bindings created for the tags using the `tag bind` subcommand. The bindings for tags are invoked before any of the bindings for the widget as a whole.

24.5 Virtual Events

The text widget generates two virtual events. You can create bindings for these events to handle them in any way you like. Often they are used to dynamically enable or disable other interface features to behave in a context-sensitive fashion.

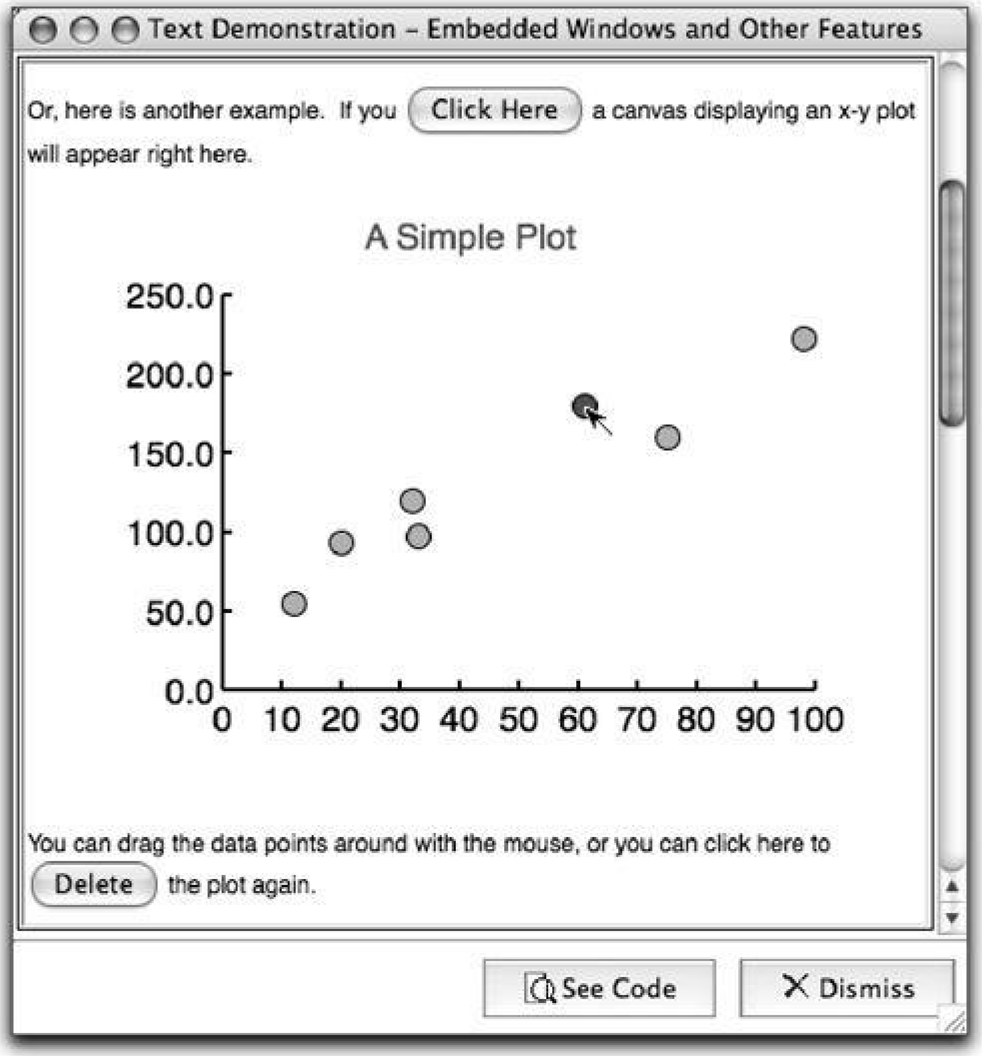
The `<<Modifed>>` event is associated with the undo mechanism described in Section 24.8. This feature requires the `-undo` option to be set to true (`1`) to be enabled. The `<<Modified>>` event is generated the first time a change is made to the text widget's content after the widget's modified flag is cleared. The event is also generated when this flag is cleared, either by the `edit undo` widget command or the explicit `edit modified` command. This event is often used to enable a "save" feature only if the widget's contents have been modified since the last save operation. An example similar to this is shown in Section 24.8.

The `<<Selection>>` event is generated whenever the widget's selection changes. It is often used to enable interface features like cut and copy only if the `sel` tag is applied to any characters, and to disable the features otherwise. An example of this is shown in Chapter 22.

24.6 Embedded Windows

In addition to text, a text widget can contain any number of widgets. These are called *embedded windows*, and can be instances of any widget class. Figure 24.3 shows the Tk widget demo embedded window example.

Figure 24.3 Embedded button and canvas widgets in a text widget



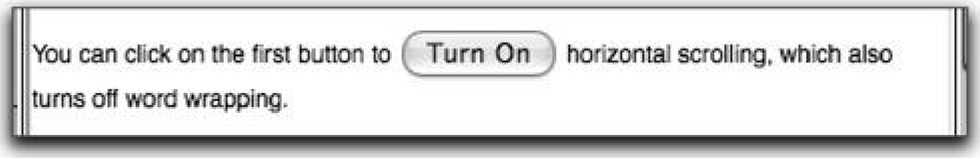
Or, here is another example. If you Click Here a canvas displaying an x-y plot will appear right here.

You can drag the data points around with the mouse, or you can click here to Delete the plot again.

Widgets are inserted inline with the text and are subject to the same padding, spacing, justification, and wrapping rules that apply to the text as if each widget were a single character, albeit a very large character.
The text widget is acting as a geometry manager for the embedded windows, so the widgets should be created as children of the text widget. Embedded windows are inserted into a text widget using the `window create` subcommand. The following code shows how this is done, and the result is shown in Figure 24.4:

```
$t insert end "You can click on the first button to "
button $t.on -text "Turn On" -command "textWindOn $w"
$t window create end -window $t.on
$t insert end " horizontal scrolling, which also turns"
$t insert end " off word wrapping."
```

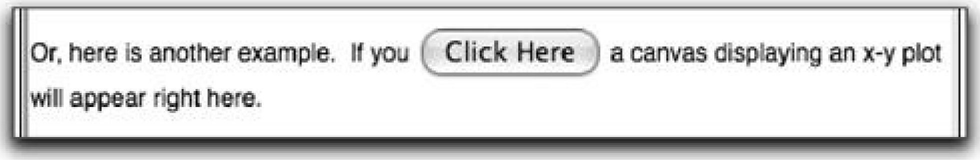
Figure 24.4 Embedding an existing button into a text widget


You can click on the first button to Turn On horizontal scrolling, which also turns off word wrapping.

Another way to insert widgets is to use the `-create` option instead of the `-window` option. The `-window` option requires that the widget be constructed separately. The `-create` option is supplied a script that is evaluated by the text widget when it needs the embedded window. Since embedded windows cannot be shared between peer text widgets, it is necessary to create unique widgets for each peer. The create script is typically used to accomplish this. The following code shows how this is done, and the result is shown in Figure 24.5:

```
$t insert end "Or, here is another example. If you "
$t window create end -create {
    button %W.click -text "Click Here" \
        -command "textWindPlot %W"
}
$t insert end " a canvas displaying an x-y plot will"
$t insert end " appear right here."
```

Figure 24.5 Using a create script to create an embedded button

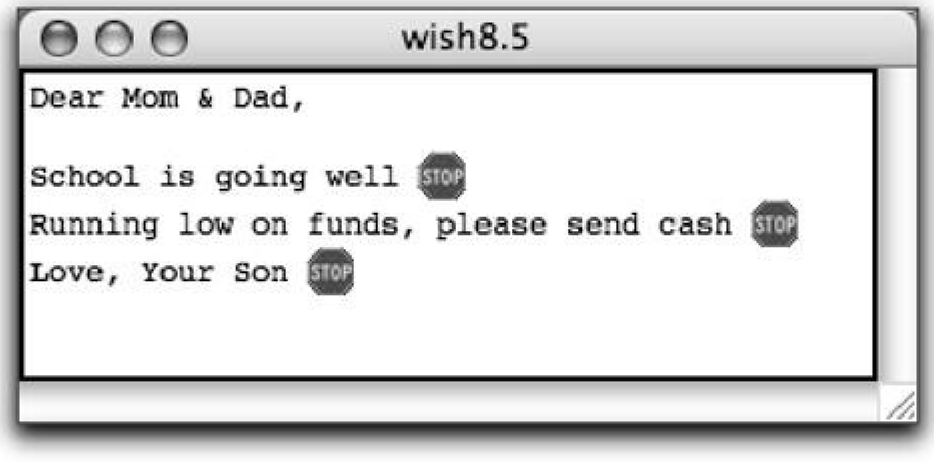

Or, here is another example. If you Click Here a canvas displaying an x-y plot will appear right here.

24.7 Embedded Images

Another type of item that can be added to a text window is an image. Images can be inserted into the text using the `image create` widget command and can appear inline with the text. This can be useful for inserting special-purpose markers, emoticons, or even photographs. The following script demonstrates inserting a specialized stop sign image used as a marker in the text; the result is shown in Figure 24.6. Notice that instead of deleting the

`<STOP>` text and replacing it with the image, a tag is applied instead with the `-elide` option set to true. This makes it possible to retrieve the original text with the `get` subcommand, or through the user doing a copy-and-paste operation involving the text.

Figure 24.6 Example of using a stop sign as an embedded image



```
Dear Mom & Dad,

School is going well
Running low on funds, please send cash
Love, Your Son
```

```
text .edit -yscrollcommand ".vsb set" \
    -xscrollcommand ".hsb set" -wrap none
scrollbar .vsb -command ".edit yview" -orient vertical
scrollbar .hsb -command ".edit xview" -orient horizontal

set letter {Dear Mom & Dad,

School is going well <STOP>
Running low on funds, please send cash <STOP>
Love, Your Son <STOP>
}

.edit insert end $letter

set stopImage [image create photo \
    -file stop_sign.gif]

.edit tag configure symbol -elide 1

proc insertStops {w} {
    foreach ix [$w search -all -forward "<STOP>" 1.0] {
        $w tag add $ix "$ix + 6 char"
        $w image create $ix -image $::stopImage
    }
}

grid .edit -row 0 -column 0 -sticky nsew
grid .vsb -row 0 -column 1 -sticky ns
grid .hsb -row 1 -column 0 -sticky ew
grid rowconfigure . 0 -weight 1
grid columnconfigure . 0 -weight 1

insertStops .edit
```

Images are subject to the same wrapping and justification rules as regular text. In the following example, nothing but images are placed in the text widget, and the text widget is used like a geometry manager to lay out the photos in a reasonably nice fashion:

```
text .t -yscrollcommand ".vsb set"
scrollbar .vsb -command ".t yview"

grid .t   -row 0 -column 0 -sticky nsew
grid .vsb -row 0 -column 1 -sticky ns
grid rowconfigure . 0 -weight 1
grid columnconfigure . 0 -weight 1

foreach f [lsort -dictionary [glob *.gif]] {
    set img [image create photo -file $f]
    .t image create end -image $img -padx 2 -pady 2
    .t insert end " "
}
```

24.8 Undo

The text widget has an unlimited undo/redo mechanism that is enabled by setting the widget's `-undo` option to `1`. The mechanism records each insert and delete action on a stack. These actions can be reversed by issuing the `edit undo` subcommand. Boundaries are inserted between edit actions to define groups of actions. These groups are used to define compound edit actions that can be undone in a single step. The `-autoseparators` option, when enabled, automatically inserts boundaries to group sequences of inserts or deletes. You can add boundaries to the stack explicitly using the `edit separator` subcommand. This can be useful if you want greater control of the undo granularity in your application.
The text widget also keeps track of modifications by way of the *modify flag* and the `<<Modified>>` virtual event. The modify flag is set to true and the virtual event is triggered after any change (insert or delete) is made to the data since the last time the flag was cleared. This flag is useful for determining if the widget's content needs to be saved, or for enabling or disabling buttons, like a save button or undo button. The flag can be read and set using the `edit modified` widget command.
The following code adds undo and redo buttons to the example from earlier in the chapter:

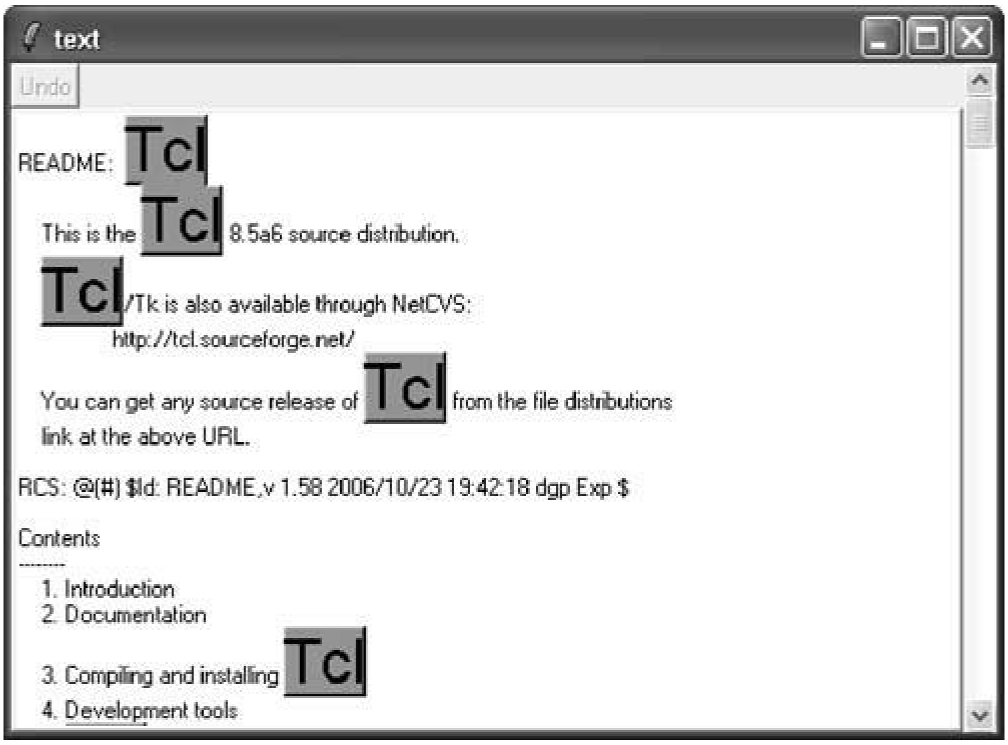
```
# text: read a file into a text widget, with undo

text .text -relief raised -bd 2 \
    -yscrollcommand ".scroll set" \
    -undo 1
scrollbar .scroll -command ".text yview"

proc loadFile file {
    .text delete 1.0 end
    set f [open $file]
    .text insert end [read $f]
    close $f

    # Mark as not being changed
    .text edit modified 0
}

proc undoText {tw} {
    catch {$tw edit undo}
}



proc undoText {tw} {
    catch {$tw edit undo}
}

proc dataModified {tw} {
    if {[$tw edit modified]} {
        .undo configure -state normal
    } else {
        .undo configure -state disabled
    }
}

frame .toolbar
button .undo -text "Undo" -command {undoText .text}

bind .text <<Modified>> {
    dataModified .text
}

pack .undo -in .toolbar -side left
pack .toolbar -side top -expand 1 -fill x
pack .scroll -side right -fill y
pack .text -side left

loadFile README
```

Notice that we modified the `loadFile` function to clear the modified flag after loading a file by adding the line

```
.text edit modified 0
```

In addition, we added the `-undo 1` option to the text widget command and packed the widgets in the proper order. Figure 24.7 shows the resulting window when the application is first launched.

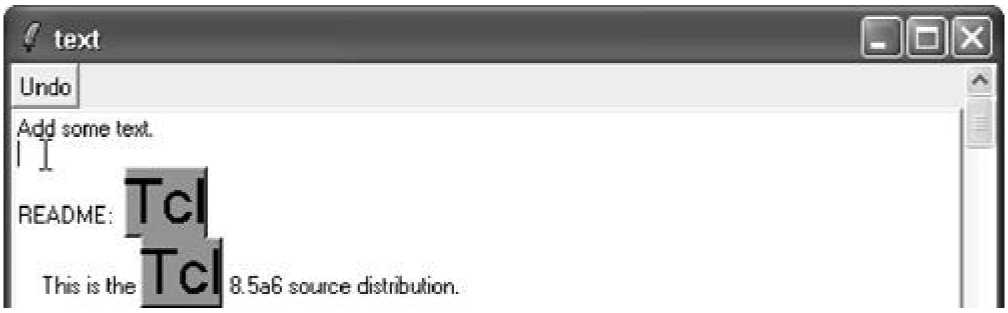
Figure 24.7 Updated text window with Undo button



```
README: Tcl
  This is the Tcl 8.5a6 source distribution.
  Tcl/Tk is also available through NetCVS:
      http://tcl.sourceforge.net/
  You can get any source release of Tcl from the file distributions
  link at the above URL.
RCS: @(#) $Id: README,v 1.58 2006/10/23 19:42:18 dgp Exp $
Contents
--------
  1. Introduction
  2. Documentation
  3. Compiling and installing Tcl
  4. Development tools
```

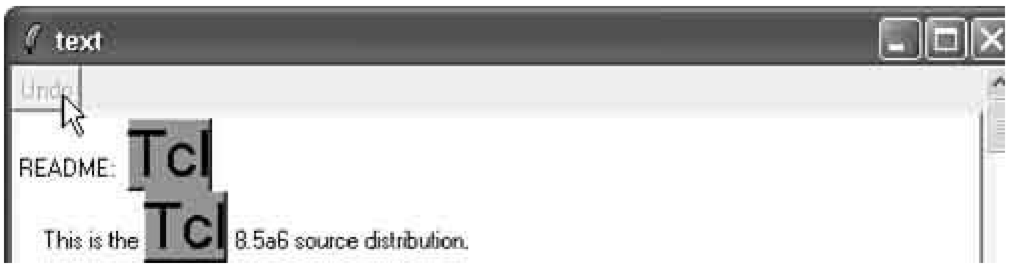
With these additions, the moment a change is made in the text widget, by typing or deleting something, the Undo button is enabled, as shown in Figure 24.8. You can then reverse the edits simply by pressing the Undo button. Once the undo stack is exhausted, the Undo button is disabled automatically, as shown in Figure 24.9.

Figure 24.8 An edit automatically enables the Undo button.



```
Add some text.

README: Tcl
  This is the Tcl 8.5a6 source distribution.
```

Figure 24.9 Exhausting the undo stack disables the Undo button.

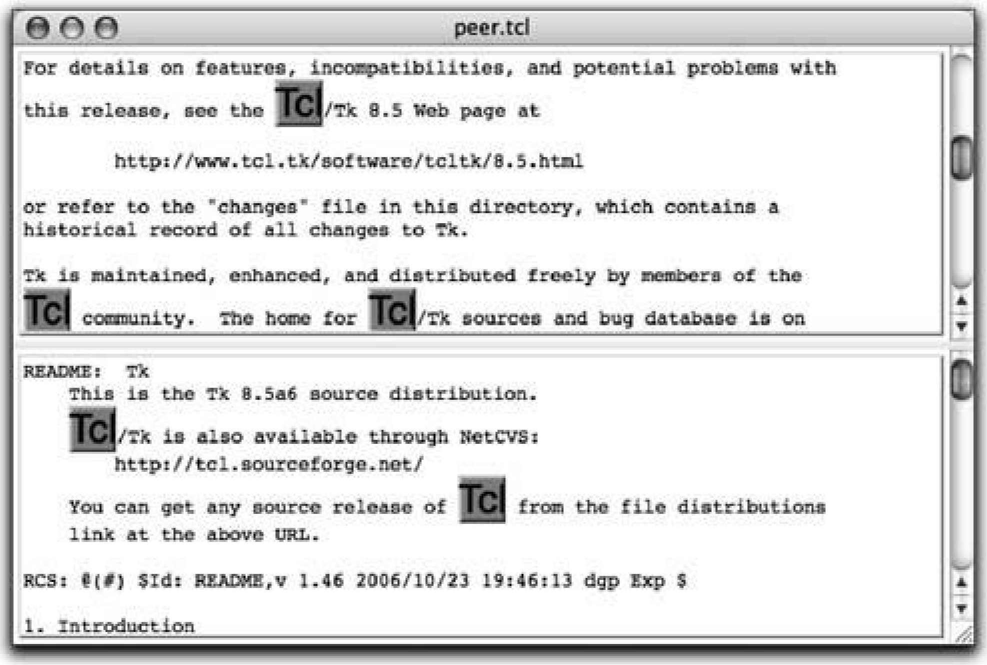


24.9 Peer Text Widgets

The `peer` text widget command is used to create duplicate text widgets that share the same text data. Any changes made in one text widget immediately appear in its peer widgets. This command is useful for creating split views, common in most modern text editors. Peer widgets share all text, images, and tags. They do not share embedded windows, the tag `sel`, or the marks `insert` and `current`. The following script shows an example of creating two text widgets as peers. Any change you make in one is reflected immediately in the other, as is shown in Figure 24.10:

```
panedwindow .pane -orient vertical
frame .tf ;# Top Frame
frame .bf ;# Bottom Frame
text .text -relief raised -bd 2 \
    -yscrollcommand ".scroll set" \
    -undo true -autoseparators true \
    -height 12
.text peer create .text2 -relief raised -bd 2 \
    -yscrollcommand ".scroll2 set" \
    -undo true -autoseparators true \
    -height 12
scrollbar .scroll -command ".text yview"
scrollbar .scroll2 -command ".text2 yview"
pack .scroll -in .tf -side right -fill y
pack .text -in .tf -side left -fill both -expand yes
pack .scroll2 -in .bf -side right -fill y
pack .text2 -in .bf -side left -fill both -expand yes
.pane add .tf -sticky nsew
.pane add .bf -sticky nsew
pack .pane -side top -fill both -expand 1
loadFile README
```

Figure 24.10 A split-pane view using peer text widgets



```
For details on features, incompatibilities, and potential problems with
this release, see the Tcl/Tk 8.5 Web page at

        http://www.tcl.tk/software/tcltk/8.5.html

or refer to the "changes" file in this directory, which contains a
historical record of all changes to Tk.

Tk is maintained, enhanced, and distributed freely by members of the
Tcl community.  The home for Tcl/Tk sources and bug database is on
```

```
README:   Tk
    This is the Tk 8.5a6 source distribution.
    Tcl/Tk is also available through NetCVS:
        http://tcl.sourceforge.net/

    You can get any source release of Tcl from the file distributions
    link at the above URL.

RCS: @(#) $Id: README,v 1.46 2006/10/23 19:46:13 dgp Exp $

1. Introduction
```

Peer text widgets are true peers. This means that destroying the original text widget does not affect the existing peer widgets. The text widget content is not destroyed until all the peer text widgets are destroyed. Peers can also create other peers. The `peer names` subcommand can be used to find the widget path names of all the peer widgets for a given text widget.

25. Selection and the Clipboard

The *selection* is a mechanism for passing information between widgets and applications. The user first selects one or more objects in a widget, for example, by dragging the mouse across a range of text or clicking on a graphical object. Once a selection has been made, the user can invoke commands in other widgets to retrieve information about the selection, such as the characters in the selected range or the name of the file containing the selection. In today's windowing environments there are two different selection models, the *selection owner model* and the *clipboard model*.

The selection owner model is used in the X Window System. There is a single owner of the selection at any given time. The widget containing the selection and the widget requesting it can be in the same or different applications. The receiving application or action can query the selection owner and obtain the information about the selection.

In the clipboard model, which is the model used in Microsoft Windows and Mac OS X as well as the X Window System, a global, virtual storage device is used called a clipboard. Selected items are copied to the clipboard, where they can be retrieved later by another widget or application. This model uses selection also, but here the ownership is confined to the current active window.

The selection, like the clipboard, is most commonly used to copy information from one place to another. Tk widgets have default bindings to do these common selection and clipboard operations, so you rarely need to manage the selection or clipboard explicitly in your applications. However, if your application needs to provide custom selection or clipboard handling, you can implement this functionality with the `selection` and `clipboard` commands. The `selection` command is used to manage the selection ownership and provide access to selection data. The `clipboard` command manages the clipboard, though the `selection` command can be used for that purpose as well.

25.1 Commands Presented in This Chapter

This chapter discusses the following commands for managing the selection:

- `selection clear ?-displayof` *`window`*`? ?-selection` *`selection`*`?`

Clears or "unselects" the current selection. `window` defaults to `.` and `selection` defaults to `PRIMARY`.

- `selection get ?-displayof window? ?-selection selection?`
  `?-type type?`

Retrieves the selection according to `selection` and `type`. `window` defaults to `.`, `selection` defaults to `PRIMARY`, and `type` defaults to `STRING`.

- `selection handle ?-selection selection? ?-type type?`
  `?-format format? window script`

Creates a handler for selection requests. `script` is evaluated whenever the `selection` is owned by `window` and someone attempts to retrieve it in the form given by `type`. `selection` defaults to `PRIMARY`, and `format` and `type` default to `STRING`. See the text and the reference documentation for more information.

- `selection own ?-displayof window? ?-selection selection?`

Returns the current owner of `selection`, or an empty string if no window owns `selection`. `window` defaults to `.` and `selection` defaults to `PRIMARY`.

- `selection own ?-command script? ?-selection selection?`
  `window`

Claims ownership of `selection` for `window`. If `script` is specified, it is executed when `window` loses the selection. `selection` defaults to `PRIMARY`.

- `clipboard clear ?-displayof window?`

Claims ownership of the clipboard on `window`'s display and removes any previous contents. `window` defaults to `.`.

- `clipboard append ?-displayof window? ?-format format?`
  `?-type type? ?--? data`

Appends `data` to the clipboard on `window`'s display in the form given by `type` with the representation given by `format` and claims ownership of the clipboard on `window`'s display. `window` defaults to `.`, and `format` and `type` default to `STRING`.

- `clipboard get ?-displayof window? ?-type type?`

Retrieves data from the clipboard on `window`'s display. `type` specifies the form in which the data is to be returned. `window` defaults to `.` and `type` defaults to `STRING`.

25.2 Selections, Retrievals, and Types

In the selection owner model, when a user selects information in a window, the window acquires selection ownership. By convention, the `PRIMARY` selection is used, though other selections are supported under the X Window System. It is possible for multiple disjointed objects to be selected simultaneously within a widget (for example, three different items in a listbox or several different polygons in a drawing window), but usually the selection consists of a single object or a range of adjacent objects. When a window acquires ownership of the selection, any previous owner is notified that it has lost ownership of the selection. At any time, the selection owner may receive a request for the selection content in a particular format, and so the selection owner must arrange to service these requests until it loses ownership of the selection.

In the clipboard model, the user selects some information in a window, and then explicitly requests that the data be copied to the clipboard. Clients can then use the `CLIPBOARD` selection to request a copy of the data from the clipboard.

When you retrieve the selection, you can ask for several different kinds of information, referred to as retrieval *types*, which are also known as *targets*. The most common type is `STRING`. In this case the contents of the selection are returned as a string. For example, if text is selected, a retrieval with type `STRING` returns the contents of the selected text; if graphics are selected, a retrieval with type `STRING` could return some string representation for the selected graphics. If the selection is retrieved with type `FILE_NAME`, the return value could be the name of the file associated with the selection. If type `LINE` is used, the return value could be the number of the selected line within its file. There are many types with well-defined meanings; refer to the X Consortium Standard Inter-Client Communication Conventions Manual (ICCCM) for more information.

The command `selection get` retrieves the selection. The type may be specified explicitly, or it may be left unspecified, in which case it defaults to `STRING`. For example, the following commands might be invoked when the selection consists of a few words on one line of a file containing the text of Shakespeare's *Romeo and Juliet*:

```
  selection get
⇒ star-crossed lovers
  selection get -type FILE_NAME
⇒ romeoJuliet
  selection get -type LINE
⇒ 6
```

These commands could be issued in any Tk application on the display containing the selection; they need not be issued in the application containing the selection.

Not every widget supports every possible selection type. For example, if the information in a widget isn't associated with a file, the `FILE_NAME` type may not be supported. If you try to retrieve the selection with an unsupported type, an error is returned. It is up to the selection owner to define the types supported for the selection and to perform the translation. Fortunately, every widget is supposed to support retrievals with type TARGETS; such retrievals return a list of all the target forms supported by the current selection owner. You can use the result of a TARGETS retrieval to pick the most convenient available type. For example, the following procedure retrieves the selection as PostScript if possible and as an unformatted string otherwise:

```
proc getSelection {} {
    set types [selection get -type TARGETS]
    if {"POSTSCRIPT" in $types} {
        return [selection get -type POSTSCRIPT]
    } else {
        return [selection get -type STRING]
    }
}
```

Tk widgets always support the type STRING. Tk widgets also support the following ICCCM types: MULTIPLE, TARGETS, TIMESTAMP, TK_APPLICATION, and TK_WINDOW, and on X Window Systems only, UTF8_STRING. MULTIPLE allows for the selection to include more than one item or block of text at a time. TARGETS gives the list of supported types. TIMESTAMP indicates when the selection occurred. The two TK types are specific to Tk applications and are discussed in the next section.

25.3 Locating and Clearing the Selection

Tk provides two mechanisms for finding out who owns the selection. The command `selection own` (with no additional arguments) checks whether the selection is owned by a widget in the invoking application. If so, it returns the path name of that widget; if there is no selection or it is owned by some other application, `selection own` returns an empty string.
The second way to locate the selection is with the retrieval types TK_APPLICATION and TK_WINDOW. These types are both implemented by Tk and are available whenever the selection is in a Tk application. The command

```
selection get -type TK_APPLICATION
```

returns the name of the Tk application that owns the selection (in a form suitable for use with the `send` command, for example), and

```
selection get -type TK_WINDOW
```

returns the path name of the widget that owns the selection. If the application that owns the selection isn't based on Tk, it does not support the `TK_APPLICATION` and `TK_WINDOW` types, and the `selection get` command returns an error. These commands also return errors if there is no selection.

The `selection clear` command clears out any selection on the display of the application's main window. It works regardless of whether the selection is in the invoking application or some other application on the same display. The following script clears out the selection only if it is in the invoking application:

```
if {[selection own] ne ""} {
    selection clear [selection own]
}
```

25.4 Supplying the Selection with Tcl Scripts

The standard widgets such as entries and texts already contain C code that supplies the selection, so you don't usually have to worry about it when writing Tcl scripts. However, it is possible to write Tcl scripts that supply the selection. For example, you might want a widget to support an additional type, such as `FILE_NAME`, for better integration with other applications in your environment.

The protocol for supplying the selection has three parts:

1. A widget must claim ownership of the selection. This deselects any previous selection, then typically redisplays the newly selected material in a highlighted fashion.
2. The selection owner must respond to retrieval requests by other widgets and applications.
3. The owner may request that it be notified when it is deselected. Widgets typically respond to deselection by eliminating the highlights on the display.

The next paragraphs describe two scenarios. The first scenario just adds a new type to a widget that already has selection support, so it deals only with the second part of the protocol. The second scenario implements complete selection support for a group of widgets that didn't previously have any; it deals with all three parts of the protocol.

Suppose that you wish to add a new type to those supported for a particular widget. For example, text widgets contain built-in support for the STRING type, but they don't automatically support the FILE_NAME type. You could add support for FILE_NAME retrievals with the following script:

```
selection handle -type FILE_NAME .t MyApp::getFile
proc MyApp::getFile {offset maxBytes} {
    variable fileName
    set last [expr {$offset + $maxBytes - 1}]
    return [string range $fileName $offset $last]
}
```

This code assumes that the text widget is named .t and that the name of its associated file is stored in a namespace variable MyApp::fileName. The selection handle command tells Tk to invoke MyApp::getFile whenever .t owns the selection and someone attempts to retrieve it with type FILE_NAME. When such a retrieval occurs, Tk takes the specified command (MyApp::getFile in this case), appends two additional numerical arguments, and invokes the resulting string as a Tcl command. In this example a command such as

```
MyApp::getFile 0 4000
```

results. The additional arguments identify a subrange of the selection by its first byte and maximum length, and the command must return this portion of the selection. If the requested range extends beyond the end of the selection, the command should return everything from the given starting point up to the end of the selection. Tk takes care of returning the information to the application that requested it. In most cases the entire selection is retrieved in one invocation of the command; for very large selections, however, Tk makes several separate invocations so that it can transmit the selection back to the requester in manageable pieces.

In the preceding example a new type was simply added to a widget that already provided some built-in selection support. If selection support is being added to a widget that has no built-in support at all, additional Tcl code is needed to claim ownership of the selection and to respond to deselect notifications. For example, consider a group of three radiobuttons named .a, .b, and .c that have already been configured with their -variable and -value options to store information about the selected button in a variable named MyApp::state. Suppose that you want to tie the radiobuttons to the selection, so that: (1) whenever a button becomes selected, it claims the X selection; (2) selection retrievals return the contents of MyApp::state; and (3) when some other widget claims the selection away from the buttons, MyApp::state is cleared and all the buttons become deselected. The following code implements these features:

```
selection handle -type STRING .a MyApp::getValue
proc MyApp::getValue {offset maxBytes} {
    variable state
    set last [expr {$offset + $maxBytes - 1}]
    return [string range $state $offset $last]
}
foreach w {.a .b .c} {
    $w config -command {
        selection own -command MyApp::selGone .a
    }
}
proc MyApp::selGone {} {
    variable state
    set state {}
}
```

The `selection handle` command and the `MyApp::getValue` procedure are similar to the previous example: they respond to `STRING` selection requests for `.a` by returning the contents of the `MyApp::state` variable. The `foreach` loop specifies a `-command` option for each of the widgets. This causes the `selection own` command to be invoked whenever the user clicks on any of the radiobuttons, and the `selection own` command claims ownership of the selection for widget `.a` (`.a` officially owns the selection regardless of which radiobutton the user selects, and it returns the value of `MyApp::state` in response to selection requests). The `selection own` command also specifies that the procedure `MyApp::selGone` should be invoked whenever the selection is claimed away by some other widget. `MyApp::selGone` sets `MyApp::state` to an empty string. All of the radiobuttons monitor `MyApp::state` for changes, so when it gets cleared, the radiobuttons deselect themselves.

25.5 The `clipboard` Command

The `clipboard` command interfaces with the windowing system clipboard. Data placed in the clipboard can be retrieved via the `selection` command's `-selection CLIPBOARD` option or the `clipboard get` command. Unlike what happens with the `PRIMARY` selection, data placed on the clipboard can be retrieved at a later time, independent of who owns the selection or if anything is selected at all.

The clipboard is typically used to implement cut, copy, and paste operations for a window. The text widget provides a binding for the virtual `<<Copy>>`

event that looks like this:

```
proc tk_textCopy {w} {
  if {![catch {set data [$w get sel.first sel.last]}]} {
    clipboard clear -displayof $w
    clipboard append -displayof $w $data
  }
}
```

This function retrieves the selection from the text widget `w`, clears the current clipboard contents, and then copies the data to the clipboard with the `clipboard append` command. The paste function performs the inverse operation:

```
proc tk_textPaste {w} {
  if {![catch {clipboard get -display $w} sel]} {
    $w insert insert $sel
  }
}
```

Note

This is a simplified version of the actual function for illustration only. The actual function also deletes the current selection in the target window and manages the edit history for the widget's undo functionality.

25.6 Drag and Drop

Drag and Drop (DND) is the ability to select an object with the mouse and "drag" the object to a destination window. This action simplifies the common copy/paste or cut/paste actions in an application. Tk does not directly support this style of interaction, but intra-application DND is easily implemented using the `selection` or `clipboard` command. The steps required for DND are as follows:

1. On `<ButtonPress-1>`, mark the current window and location, assuming the mouse cursor is over an already selected object.
2. On mouse motion, don't start the "drag" operation until the mouse moves sufficiently from the starting point. Once the mouse has

moved far enough to indicate a “drag,” initiate the drag and provide feedback, for example, by changing the cursor.

3. On `<ButtonRelease-1>`, use the x- and y-coordinates relative to the display root to identify the window under the mouse (the `winfo containing` command can do this).
4. Get the selection and insert the data at the location under the mouse as indicated by the `%x` and `%y` location in the target window.

This simple approach works fine inside a Tk application, but what if you want to support DND between two applications, or between the application and the windowing system? There are extensions that implement standard DND protocols across multiple platforms, such as `TkDND` (see http://www.sourceforge.net/projects/tkdnd). This extension handles the sequence of operations for you so that the only code the application needs to provide is the code to perform the actual extraction of the selection and the insertion at the drop point.

26. Window Managers

A *window manager* is a component of the windowing system whose main function is to control the arrangement and decoration of all the toplevel windows on each screen. In this respect it is similar to a geometry manager, except that instead of managing internal windows within an application, it manages the toplevel windows of all applications. The window manager allows each application to request particular locations and sizes for its toplevel windows, which users can override interactively. Window managers also serve several purposes besides geometry management: they add decorative frames around toplevel windows; they allow windows to be iconified and deiconified; and they notify applications of certain events, such as user requests to close the window.

Some operating systems have just one desktop environment or window manager such as Microsoft Windows and Apple's Mac OS X Aqua, whereas Linux and other Unix-based operating systems allow for the existence of many different window managers that implement different styles of layout, provide different kinds of decoration and icon management, and so on. Some examples include the GNOME and KDE desktop environments, along with the older CDE, or Common Desktop Environment. Only a single window manager runs on a display at any given time, and the user gets to choose which one.

With Tk you use the `wm` command to communicate with the window manager. Tk implements the `wm` command so that any Tk-based application should work with any window manager without regard to the type of window manager or desktop environment currently in use. In order to create a well-behaved application on the desktop, your programs need to call the `wm` command to properly inform the window manager about window titles, icons, and so on.

Note

In general, an application can only send requests or provide hints to the window manager. The window manager is free to ignore or modify those requests and hints. For example, an application can request a certain size for its toplevel, but the window manager might enforce a smaller size because of the available dimensions of the display, or the application might request a feature not implemented by the particular window manager.

26.1 Commands Presented in This Chapter

This chapter discusses the `wm` command for interacting with the window manager. In all of the following commands, *`window`* must be the name of a toplevel window. Many of the commands, such as `wm aspect` or `wm group`, are used to set and query various parameters related to window management. For these commands, if the parameters are specified as null strings, the parameters are removed completely; if the parameters are omitted, the command returns the current settings for the parameters.

- `wm aspect` *`window`* `?`*`xThin yThin xFat yFat`*`?`

Constrains the aspect ratio (width/length) of *`window`* to the values given.

- `wm attribute` *`window`* `?`*`attribute`*`? ?`*`value attribute value`* `...?`

Sets or queries the window-system-specific attributes. See Section 26.8 and the reference documentation.

- `wm client` *`window`* `?`*`name`*`?`

Sets or queries the `WM_CLIENT_MACHINE` property for *`window`*, which gives the name of the machine on which *`window`*'s application is running.

- `wm command` *`window`* `?`*`value`*`?`

Sets or queries the `WM_COMMAND` property for *`window`*, which contains the command line used to initiate *`window`*'s application.

- `wm deiconify` *`window`*

Arranges for *`window`* to be displayed in normal (noniconified) fashion.

- `wm focusmodel` *`window`* `?`*`model`*`?`

Sets or queries the focus model for *`window`*. *`model`* must be `active` or `passive`.

- `wm forget` *`window`*

Arranges for *`window`* to no longer be managed as a toplevel window by the window manager.

- `wm geometry` *`window`* `?`*`value`*`?`

Sets or queries the requested geometry for *`window`*. *`value`* must have the form `=`*`width`*`x`*`height`*`±`*`x`*`±`*`y`* (any of `=`, *`width`*`x`*`height`*, or ±*`x`*±*`y`* can be omitted).

- `wm grid` *`window`* `?`*`baseWidth baseHeight widthInc heightInc`*`?`

Indicates that *`window`* is to be managed as a gridded window and also specifies the relationship between grid units and pixel units. See the reference documentation for more information.

- `wm group` *`window`* `?`*`leader`*`?`

Sets or queries the window group to which *`window`* belongs. *`leader`* must be the name of a toplevel window, or an empty string to remove *`window`* from its current group.

- `wm iconbitmap` *`window`* `?-default? ?`*`bitmap`*`?`

Sets or queries the bitmap for *`window`*'s icon. If the `-default` option is specified, the icon is applied to all toplevel windows (existing and future) to which no other specific icon has yet been applied.

- `wm iconify` *`window`*

Arranges for *`window`* to be iconified.

- `wm iconmask` *`window`* `?`*`bitmap`*`?`

Sets or queries the mask bitmap for *`window`*'s icon.

- `wm iconname` *`window`* `?`*`string`*`?`

Sets or queries the string to be displayed in *`window`*'s icon.

- `wm iconphoto` *`window`* `?-default?` *`image1`* `?`*`image2 ...`*`?`

Sets the title bar icon for *`window`* based on the named photo images. Multiple images are allowed to provide different image sizes. If the `-default` option is specified, the icon is applied to all future toplevel windows as well.

- `wm iconposition` *`window`* `?`*`x y`*`?`

Sets or queries the hints about where on the screen to display *`window`*'s icon.

- `wm iconwindow` *`window`* `?`*`icon`*`?`

Sets or queries the window to use as the icon for *`window`*. *`icon`* must be the path name of a toplevel window.

- `wm manage` *`window`*

Arranges for *`window`* to be managed as a toplevel window by the window manager.

- `wm maxsize` *`window`* `?`*`width height`*`?`

Sets or queries the maximum permissible dimensions for *`window`* during interactive resize operations.

- `wm minsize` *`window`* `?`*`width height`*`?`

Sets or queries the minimum permissible dimensions for *`window`* during interactive resize operations.

- `wm overrideredirect` *`window`* `?`*`boolean`*`?`

Sets or queries the override-redirect flag for *`window`*.

- `wm positionfrom` *`window`* `?`*`whom`*`?`

Sets or queries the source of the position specification for *`window`*. *`whom`* must be `program` or `user`.

- `wm protocol` *`window`* `?`*`protocol`*`? ?`*`script`*`?`

Arranges for *`script`* to be executed whenever the window manager sends a message to *`window`* with the given *`protocol`*. *`protocol`* must be the name of an atom for a window manager protocol, such as `WM_DELETE_WINDOW`. If *`script`* is an empty string, the current handler for *`protocol`* is deleted. If *`script`* is omitted, the current script for *`protocol`* is returned (or an empty string if there is no handler for *`protocol`*). If both *`protocol`* and *`script`* are omitted, the command returns a list of all protocols with handlers defined for *`window`*.

- `wm resizable` *`window`* `?`*`width height`*`?`

Controls whether the user may interactively change the *`width`* and the *`height`* of the *`window`*, according to the Boolean values given.

- `wm sizefrom` *`window`* `?`*`whom`*`?`

Sets or queries the source of the size specification for *`window`*. *`whom`* must be `program` or `user`.

- `wm state` *`window`* `?`*`newstate`*`?`

Sets or queries the current state of *`window`*. The state can be one of `normal`, `iconic`, `withdrawn`, or `zoomed` (Windows and Mac OS X only).

- `wm title` *`window`* `?`*`string`*`?`

Sets or queries the title string to display in the decorative border for *`window`*.

- `wm transient` *`window`* `?`*`master`*`?`

Sets or queries the transient status of *`window`*. *`master`* must be the name of a toplevel window on whose behalf *`window`* is working as a transient.

- `wm withdraw` *`window`*

Arranges for *`window`* to be withdrawn from the screen.

26.2 Window Sizes

Even if a Tk application never invokes the `wm` command, Tk still communicates with the window manager on the application's behalf so that its toplevel windows appear on the screen. By default each toplevel window appears in its *natural size*, which is the size it requested using the normal Tk mechanisms for geometry management. Tk forwards the natural size to the window manager, and most window managers honor the request. If the natural size of a toplevel window should change, Tk forwards the new size to the window manager, and the window manager resizes the window to correspond to the latest request.
If the user interactively resizes a toplevel window, the window's natural size is ignored from that point on. Regardless of how the internal needs of the window change, its size remains as set by the user. A similar effect occurs if you invoke the `wm geometry` command, as in the following example:

```
wm geometry .w 300x200
```

This command forces `.w` to be 300 pixels wide and 200 pixels high, just as if the user had resized the window interactively. The natural size for `.w` is ignored, and the size specified in the `wm geometry` command overrides any size that the user might have specified interactively (though the user can resize the window again to override the size in the `wm geometry` command).
If you would like to restore a window to its natural size, you can invoke `wm geometry` with an empty geometry string:

```
wm geometry .w {}
```

This causes Tk to forget any size specified by the user or by `wm geometry`, so the window returns to its natural size.
If you want to limit interactive resizing, you can invoke `wm minsize` and/or `wm maxsize` to specify a range of acceptable sizes. For example, the script

```
wm minsize .w 100 50
wm maxsize .w 400 150
```

allows `.w` to be resized but constrains it to be 100 to 400 pixels wide and 50 to 150 pixels high. If the command

```
wm minsize .w 1 1
```

is invoked, there is effectively no lower limit on the size of `.w`. If you set a minimum size without a maximum size, the maximum size is the size of the display; if you set a maximum size without a minimum size, the minimum size is unconstrained. You can disable or enable interactive resizing with the `wm resizable` command, which allows you to set two Boolean values to indicate whether horizontal or vertical resizing is allowed. The following command disables both horizontal and vertical resizing, fixing the window at whatever size it is when the command is executed:

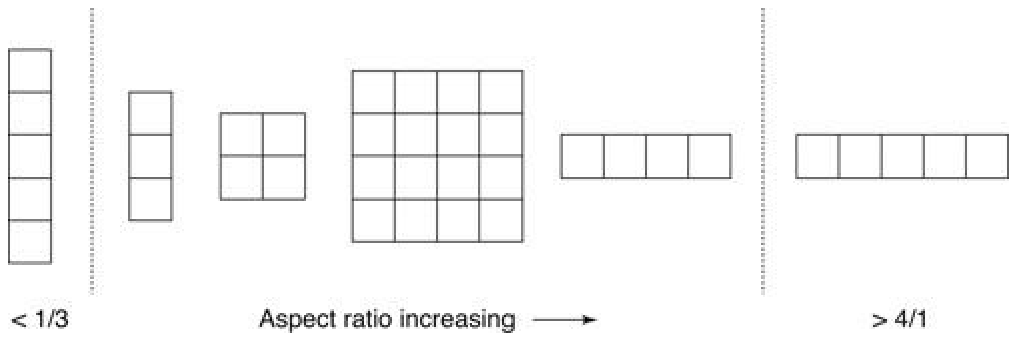
```
wm resizable .w 0 0
```

In addition to constraining the dimensions of a window, you can also constrain its aspect ratio (width divided by height) using the `wm aspect` command. For example,

```
wm aspect .w 1 3 4 1
```

tells the window manager not to let the user resize the window to an aspect ratio less than 1/3 or greater than 4. See Figure 26.1 for examples of various aspect ratios. After the command just given, the window manager allows the user to resize `.w` to any of the shapes between the two dotted lines but not to those outside the dotted lines.

Figure 26.1 The aspect ratio of a window is its width divided by its height.



26.3 Window Positions

Controlling the position of a toplevel window is simpler than controlling its size. Users can move windows interactively, and an application can also move its own windows using the `wm geometry` command. For example, the command

```
wm geometry .w +100+200
```

positions `.w` so that its upper left corner is at pixel (100,200) on the display. If either of the `+` characters is replaced with a `-`, the coordinates are measured from the right and bottom sides of the display. For example,

```
wm geometry .w -0-0
```

positions `.w` at the lower right corner of the display.

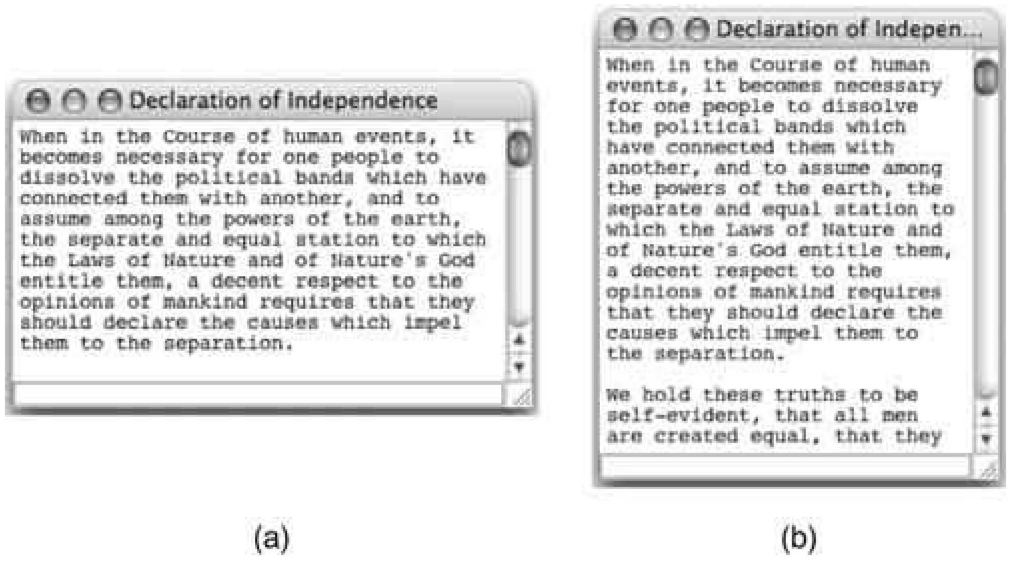
Note

If you are using a virtual root window manager, the positions you specify in a `wm geometry` command may be relative either to the screen or to the virtual root window. Consult the documentation for your window manager to see which of these is the case. With some window managers, you can use options to the window manager along with the `wm positionfrom` command to control which interpretation is used.

26.4 Gridded Windows

In some cases it doesn't make sense to resize a window to arbitrary pixel sizes. For example, consider the application shown in Figure 26.2. When the user resizes the toplevel window, the text widget changes size in response. Ideally the text widget should always contain an integral number of characters in each dimension, and sizes that result in partial characters should be rounded off.

Figure 26.2 An example of gridded geometry management



```
When in the Course of human events, it
becomes necessary for one people to
dissolve the political bands which have
connected them with another, and to
assume among the powers of the earth,
the separate and equal station to which
the Laws of Nature and of Nature's God
entitle them, a decent respect to the
opinions of mankind requires that they
should declare the causes which impel
them to the separation.
```

```
When in the Course of human
events, it becomes necessary
for one people to dissolve
the political bands which
have connected them with
another, and to assume among
the powers of the earth, the
separate and equal station to
which the Laws of Nature and
of Nature's God entitle them,
a decent respect to the
opinions of mankind requires
that they should declare the
causes which impel them to
the separation.

We hold these truths to be
self-evident, that all men
are created equal, that they
```

Gridded geometry management accomplishes this effect. When gridding is enabled for a toplevel window, the window's dimensions are constrained to lie on an imaginary grid. The geometry of the grid is determined by one of the widgets contained in the toplevel window (e.g., the text widget in Figure 26.2) so that the widget always holds an integral number of its internal objects. Usually the widget that controls the gridding is a text-oriented widget such as an entry or listbox or text. If the user interactively resizes the window from the dimensions in Figure 26.2(a) to those in Figure 26.2(b), the window manager rounds off the dimensions so that the text widget holds

an integral number of characters in each dimension.
To enable a widget to control the grid for a toplevel window, set the `-setgrid` option to `1` in the controlling widget. Only one widget can control the gridding for a given toplevel. The following code was used in the example in Figure 26.2, where the text widget is `.t`:

```
.t configure -setgrid 1
```

This command has several effects. First, it identifies to the toplevel window that this widget determines the grid size for the overall window. Second, it changes the unit of measurement for dimensions used in the `wm geometry` command. For the text widget, the unit of measurement is a character. For example, the command

```
wm geometry . 50x30
```

sets the size of the main window so that `.t` is 50 characters wide and 30 lines high. The units in `wm minsize` and `wm maxsize` as well as the `-width` and `-height` options for the widget itself are also measured in grid units instead of points.
Gridding may also be defined using the `wm grid` command. This is used when the grid unit size is determined by something other than a widget.

Note

Gridding works well only for windows with fixed-size objects, such as a text window with a monospace font. If different characters have different sizes, the window’s size won’t necessarily be an integral number of characters.
Furthermore, in order for gridding to work correctly, you must have configured the internal geometry management of the application so that the controlling window stretches and shrinks in response to changes in the size of the toplevel window, for example, by gridding it with the option `-sticky nsew`.

26.5 Window States

At any given time each toplevel window is in a given *state*. In the *normal* or *deiconified* state the window appears on the screen. In the *zoomed* state, supported on Windows and Mac OS X systems, the window appears in a maximized state on the screen. In the *iconified* state the window does not appear on the screen, but an icon is displayed in the taskbar or desktop instead. In the *withdrawn* state the window does not appear anywhere on the screen and the window is ignored completely by the window manager.

New toplevel windows start off in the normal state. You can use the facilities of your window manager to iconify a window interactively, or you can invoke the `wm iconify` command within the window's application; for example:

```
wm iconify .w
```

If you invoke `wm iconify` immediately, before the window first appears on the screen, it starts off in the iconic state. The command `wm deiconify` causes a window to revert to normal state again.

The command `wm withdraw` places a window in the withdrawn state. If invoked immediately, before a window has appeared on the screen, the window starts off withdrawn. The most common use for this command is to prevent the main window of an application from ever appearing on the screen (in some applications the main window serves no purpose and the user interface is presented with a collection of toplevel windows). Once a window has been withdrawn, it can be returned to the screen with either `wm deiconify` or `wm iconify`.

The `wm state` command sets or returns the current state for a window:

```
  wm iconify .w
  wm state .w
⇒ iconic
```

26.6 Decorations

When a window appears on the screen in the normal state, the window manager usually adds a decorative frame around it. The frame typically displays a title for the window and contains interactive controls for resizing the window, moving it, and so on. For example, Mac OS X Aqua decorated the window in Figure 26.2.

The `wm title` command allows you to set the title that is displayed in a window's decorative frame. For example, the command

```
wm title . "Declaration of Independence"
```

was used to set the title for the window in Figure 26.2.

Other `wm` commands have indirect effects on the interactive frame controls. For example, if the resizable flags are set to false (`wm resizable . 0 0`), the maximize button or the resize grab handle (lower right corner) is not displayed. The exact change of the controls depends on which window manager is in use.

The `wm` command provides several options for controlling what is displayed when a window is iconified. First, you can use `wm iconname` to specify a title to display in the icon. Second, some window managers allow you to specify an image to be displayed in the icon. The `wm iconphoto` command allows you to set this image. Some older window managers support only simple two-color bitmaps for icons. The `wm iconbitmap` and `wm iconmask` are used to set a two-color icon and transparency mask respectively. Third, some window managers allow you to use one window as the icon for another; `wm iconwindow` sets up such an arrangement if your window manager supports it. Finally, you can specify a position on the screen for the icon with the `wm iconposition` command.

Note

Almost all window managers support `wm iconname`. The `wm iconphoto` command is handled differently on different window managers, fewer support `wm iconbitmap`, and almost no window managers support `wm iconwindow` very well. You need to experiment with these commands to determine which ones meet your requirements based on the platform or platforms your application needs to support.

26.7 Special Handling: Transients, Groups, and Override-Redirect

You can ask the window manager to provide special treatment for windows to implement features such as toolbars, nonmodal dialog boxes, and splash

screens.
You can mark a toplevel window as *transient* with a command like the following:

```
wm transient .w .
```

This indicates to the window manager that `.w` is a short-lived window such as a dialog box, working on behalf of the application's main window. The last argument to `wm transient` (which is `.` in the example) is referred to as the *master* for the transient window. Most window managers ensure that a transient window always remains above its master. Additionally, the window manager may treat transient windows differently from other windows by providing less decoration or by iconifying and deiconifying them whenever their master is iconified or deiconified.
In situations where several long-lived windows work together, you can use the `wm group` command to tell the window manager about the group. The following script tells the window manager that the windows `.top1`, `.top2`, `.top3`, and `.top4` are working together as a group, and `.top1` is the group *leader*:

```
foreach i {.top2 .top3 .top4} {
    wm group $i .top1
}
```

The window manager can then treat the group as a unit, and it may give special treatment to the leader. For example, when the group leader is iconified, all the other windows in the group might be removed from the display without icons being displayed for them: the leader's icon represents the whole group in this case. When the leader's icon is deiconified again, all the windows in the group might return to the display. The exact treatment of groups is up to the window manager, and different window managers may handle them differently. The leader for a group need not actually appear on the screen (e.g., it could be withdrawn).
In some rare cases it is important for a toplevel window to be completely ignored by the window manager: no decorations, no interactive manipulation of the window via the window manager, no iconifying, and so on. Typical examples of such a window are a hover-help window or a splash screen. In these cases, the windows should be marked as *override-redirect* using a command similar to the following:

```
wm overrideredirect .splash true
```

This command must be invoked before the window has actually appeared on the screen.

Note

Menus automatically mark themselves as override-redirect so you don't need to do this for them.

26.8 System-Specific Window Attributes

Each windowing system has unique features that do not have any portable equivalent in other windowing systems. The `wm attribute` command is used to query and set some of these unique features:

wm attribute *window* ?*attribute*? ?*value attribute value* ...?

Currently, all windowing systems support the following attributes:

- `-fullscreen` *boolean*

When true, the window fills the entire screen without any window manager decoration. This feature is usually used in conjunction with `-topmost`.

- `-topmost` *boolean*

When true, the window always appears above all others.
On Windows, the following additional attributes are available:

- `-alpha` *number*

Controls the transparency of the toplevel. Valid values are in the range `0.0` (transparent) to `1.0` (opaque).

- `-disabled` *boolean*

When true, the window is in a disabled state and cannot take focus.

- `-toolwindow` *boolean*

When true, causes the window to take on the style of a tool window.

- `-transparentcolor` *color*

Specifies the transparent color of the toplevel. If the empty string is specified (default), no transparent color is used.
On Mac OS X Aqua, the following additional attributes are available:

- `-alpha` *number*

Controls the transparency of the toplevel. Valid values are in the range `0.0` (transparent) to `1.0` (opaque).

- `-modified` *boolean*

When true, the window close icon shows the modified state. This is typically used to show that the application has modified, unsaved data.

- `-notify` *boolean*

When true, it causes the icon in the dockbar to bounce and, if enabled, the voice announcement is triggered to indicate that the application needs attention.

- `-titlepath` *pathname*

When set, specifies the path of the file referenced by the icon in the title bar of the window, which can be dragged and dropped in lieu of the file's Finder icon.
On X11, the following additional attributes are available:

- `-zoomed` *boolean*

When true, maximizes the window as with `wm state` of `zoomed` for Windows and Mac OS X.

26.9 Dockable Windows

The `wm manage` command is used to make any window into a toplevel window managed by the window manager, and the `wm forget` command performs the inverse operation, informing the window manager to stop managing the window. Although this command can work with any widget, it is ideally suited for use with the toplevel and frame widgets. These commands are used to effect a "dock" operation, where two toplevel windows are joined to make a single toplevel window, and an "undock" operation, where a subwindow is removed from a toplevel window and made into a separate toplevel window. These commands alone do not perform the complete operation but when combined with another geometry manager command complete the task. A typical dock operation would be

```
wm forget .top.frame1
grid.top.frame1 -row 0 -column 1
```

An undock operation would be

```
grid forget .top.frame1
wm manage .top.frame1
```

There are some limitations to be aware of when using the `wm manage` command:

- When "docking," the widget must be a child of the geometry manager's master widget.
- Toplevel class windows behave like frames when not managed by the window manager, so only the frame properties of the widget apply. This means the `-menu` setting is ignored and the menu bar is no longer available in the window. It does reappear when the toplevel window is again managed by the window manager.
- Only a toplevel widget can have a menu bar because this is the only Tk widget with a `-menu` option.
- Bindtags that have been set explicitly in any subwindow of a docked or undocked window must be reset or updated. This is because the bindtag contains the widget path of the toplevel window. Since the path of the toplevel window has been changed as a result of the operation, the old bindtag is no longer appropriate. (See Section 22.8 for more information on bindtags.)

26.10 Window Close

Closing a toplevel window can be performed by the Tcl script using the `destroy` command, or by the window manager when the user presses the close button on the decoration frame. There are times when it is necessary for the application to be notified when the window manager closes a window so that additional actions may be taken or the operation prevented altogether. This operation can be intercepted with the `wm protocol` command. There are several *window manager protocols* that are implemented in X11, but the only one relevant here and uniformly recognized on all platforms and window managers is the `WM_DELETE_WINDOW` protocol. The window manager invokes this protocol when it wants the application to destroy the window, as when the user asks the window manager to close the window.
The `wm protocol` command arranges to execute a script whenever a particular protocol is triggered. For example, the command

```
wm protocol . WM_DELETE_WINDOW {
  set response [tk_messageBox -type yesno \
                -message "Really quit?"]
  if {$response eq "yes"} {
    destroy .
  }
}
```

prompts the user for confirmation to quit whenever the window manager asks the application to close its main window. If the user selects “No,” the window is not actually destroyed. If you’ve not used the `wm protocol` command to register a handler for the `WM_DELETE_WINDOW` protocol on a toplevel, Tk takes the default action of destroying the window. For other protocols, nothing happens unless you specify an explicit handler.

26.11 Session Management

Some window managers manage the user’s session, recording which applications are running when the session is closed (i.e., when the user logs off the system) and starting those same applications when a new session is started (i.e., when the user logs on to the system). Two special `wm` commands can provide the necessary information so that the window manager can successfully save and restore the Tk application. The `wm client` command provides the name of the host machine running the application. The `wm command` command provides a program path name with arguments (i.e., a full command line) that the session manager can use to restart the application. For example,

```
wm client . sprite.berkeley.edu
wm command . {browse /usr/local/bin}
```

indicates that the application is running on the machine `sprite.berkeley.edu` and was invoked with the shell command `browse/usr/local/bin`.

27. Focus, Modal Interaction, and Custom Dialogs

This chapter discusses the management of the *input focus window* and *modal windows*. Input focus determines which widget in your application receives keyboard events. A modal window is a child window that requires users to interact with it before they can continue interacting with the main window. The most common examples of modal windows are dialogs such as a file selector.
The default Tk bindings automatically handle focus management and modal interactions in a way that most users expect. For example, clicking on an entry automatically gives it focus so that the user can enter text into it, and the dialogs provided by Tk automatically behave as modal windows. Typically the only situation where you would manage focus and modal interaction explicitly is if you need to create you own custom dialogs for your application.

27.1 Commands Presented in This Chapter

This chapter describes the following commands for use in manipulating focus, modal interaction, and custom dialogs:

- `focus`

Returns the path name of the focus window on the display containing the application's main window, or an empty string if no window in this application has the focus on that display.

- `focus -displayof` *`window`*

Returns the name of the focus window on the display containing *`window`*. If the focus window for *`window`*'s display isn't in this application, the return value is an empty string.

- `focus -lastfor` *`window`*

Returns the name of the most recent window to have the input focus among all the windows in the same toplevel as *`window`*. If no window in that toplevel has ever had the input focus, or if the most recent focus window has been deleted, the name of the toplevel is returned.

- `focus ?-force?` *`window`*

Sets the application's focus window to `window`. The `-force` option also forces the application to have focus; this feature should be used sparingly.

- `tk_focusFollowsMouse`

Switches Tk to an implicit focus model, where focus is set to a window whenever the mouse enters it.

- `tk_focusNext` *`window`*

Returns the next window after `window` in the focus order.

- `tk_focusPrevious` *`window`*

Returns the window just before `window` in the focus order.

- `grab ?-global?` *`window`*

Same as the `grab set` command.

- `grab current ?`*`window`*`?`

Returns the name of the current grab window for `window`'s display, or an empty string if there is no grab for that display. If `window` is omitted, it returns a list of all windows grabbed by this application for all displays.

- `grab release` *`window`*

Releases the grab on `window`, if there is one.

- `grab set ?-global?` *`window`*

Sets a grab on `window`, releasing any previous grab on `window`'s display. If `-global` is specified, the grab is global; otherwise it is local.

- `grab status` *`window`*

Returns `none` if no grab is currently set on `window`, `local` if a local grab is set, and `global` if a global grab is set.

- `tkwait variable` *`varName`*

Waits until the variable `varName` changes value, then returns.

- `tkwait visibility` *`window`*

Waits until the visibility state of `window` changes, then returns.

- `tkwait window` *`window`*

Waits until `window` is destroyed, then returns.

27.2 Input Focus

At any given time one window of an application is designated as the *input focus window*, or *focus window* for short. All keystrokes received by the application are directed to the focus window and are processed according to its event bindings; if the focus window has no bindings for the keystroke, the keystroke is ignored.

Note

The focus window is used only for key press and key release events. Mouse-related events, such as enter, leave, button press, and button release, are always delivered to the window under the mouse, regardless of the focus window. Furthermore, the focus window determines only what happens once a keystroke event arrives at a particular application; it does not determine which of the applications on the display receives keystrokes. The choice of a focus application is made by the window manager, not Tk.

27.2.1 Focus Model: Explicit versus Implicit

There are two possible ways of handling the input focus, known as the *implicit* and *explicit* focus models. In the implicit model the focus follows the mouse: keystrokes are directed to the window under the mouse pointer, and the focus window changes implicitly when the mouse moves from one window to another. In the explicit model the focus window is set explicitly and doesn't change until it is explicitly reset; mouse motions do not automatically change the focus.
Tk implements the explicit focus model, for several reasons. First, the explicit model allows you to move the mouse cursor out of the way when you're typing in a window; with the implicit model you'd have to keep the mouse in the window while you type. Second, and more important, the explicit model allows an application to change the focus window without the user moving the mouse. For example, when an application pops up a dialog box that requires type-in, such as one that prompts for a file name, it can set the input focus to the appropriate window in the dialog without your having to move the mouse, and it can move the focus back to its original window when you're finished with the dialog box. This allows you to keep your hands on the keyboard. Similarly, when you're typing in a form, the application can move the input focus to the next entry in the form each time you press the Tab key, so that you can keep your hands on the keyboard and work more efficiently. Last, if you want an implicit focus model, it can always be achieved with event bindings that change the focus each time the mouse cursor enters a new window. Tk provides a command named `tk_focusFollowsMouse` that switches your application to an implicit focus model:

it reconfigures Tk bindings so that the focus is set to a window whenever the mouse enters it.
Tk applications don’t need to worry about the input focus very often because the default bindings for text-oriented widgets already take care of the most common situations. For example, when you click mouse button 1 over an entry or text widget, the widget automatically makes itself the focus window. As an application designer you need to set the focus only in cases like those in the previous paragraph where you want to move the focus among the windows of your application to reflect the flow of work. You should be careful when doing this, though, because most windowing systems use the explicit focus model. Changing the model can prove disconcerting to your users.

27.2.2 Setting the Input Focus

To set the input focus, invoke the `focus` command with a widget name as argument:

```
focus .dialog.entry
```

This directs subsequent keystrokes received by the application to `.dialog.entry`; the previous focus window no longer receives keystrokes. The new focus window displays some sort of highlight, such as a blinking insertion cursor, to indicate that it has the focus, and the previous focus window stops displaying its highlight.

Note

The `focus -force` option also attempts to force the application to have focus; not all window managers honor this request. In general, forcing focus to your application should be used sparingly; it is akin to going up to a person at a party and shouting, “Pay attention to me!”

27.2.3 Querying the Input Focus

Sometimes it’s useful to query the current focus window, particularly if your application needs to redirect focus temporarily to a different window. The `focus` command returns the name of the application's current focus window, or an empty string if no window in the application is currently the focus window. In contrast, `focus -lastfor` returns the name of the last focus window within the toplevel containing the specified window; this is the name of the window that receives the input focus the next time the window manager gives focus to the toplevel.

27.3 Modal Interactions

Usually the user of a Tk application has complete control over what operations to perform and in what order. The application offers a variety of panels and controls, and the user selects among them. However, there are times when it's useful to restrict the user's range of choices or force the user to do things in a certain order; these are called *modal interactions*. The best example of a modal interaction is a dialog box: the application is carrying out some function requested by the user such as writing information to a file when it discovers that it needs additional input from the user, such as the name of the file to write. It displays a dialog box and forces the user to respond to the dialog box, for example, by selecting a file name, before proceeding. After the user responds, the application completes the operation and returns to its normal mode of operation where the user can do anything he or she pleases.
Tk provides two mechanisms for use in modal interactions. First, the `grab` command allows you to restrict the user temporarily so that he or she can interact with only a few of the application's windows (for example, only the dialog box). Second, the `tkwait` command allows you to suspend the evaluation of a script until a particular event has occurred, such as the user responding to a dialog box, and then continue the script after this has happened.

Note

A word of caution is in order here. Although modal interactions are sometimes useful, most experts agree that they should be kept to a minimum. Users can become frustrated if their range of choices is constantly being limited by modes, and mode switches can be confusing.

27.3.1 Grabs

Mouse events such as button presses and mouse motions are normally delivered to the window under the pointer. However, it is possible for a window to claim ownership of the mouse so that mouse events are delivered only to that window and its descendants in the Tk window hierarchy. This is called a *grab*. When the mouse is over one of the windows in the grab subtree, mouse events are delivered and processed just as if no grab were in effect. When the mouse is outside the grab subtree, button presses and releases and mouse motion events are delivered to the grab window instead of the window under the mouse, and window entry and exit events are discarded. Thus, a grab prevents the user from interacting with windows outside the grab subtree.
The `grab` command sets and releases grabs. For example, if you've created a dialog box named `.dlg` and you want to prevent the user from interacting with any window except `.dlg` and its subwindows, you can invoke the command

```
grab set .dlg
```

After the user has responded to the dialog box, you can release the grab with the command

```
grab release .dlg
```

If the dialog box is destroyed after the user has responded to it, there's no need to invoke `grab release`: Tk releases the grab automatically when the grab window is destroyed.
To see how grabs work, try the following script:

```
button .b1 -text "Unclickable 1"
button .b2 -text "Unclickable 2"
button .b3 -text "Grabby Button" -command { destroy .b3 }
label .l -text "Text entry"
entry .e

pack .b1 .b2 .b3 .l .e
grab .b3
```

In this script, you can interact with only one widget, the button `.b3`, which

has a grab on input events. You cannot set the keyboard focus to the entry box or enter text. If you click on `.b3`, labeled “Grabby Button,” the callback script destroys the button. This finally allows you to interact with the other buttons as well as the entry widget.
The most common way to use grabs is to set a grab on a toplevel window so that only a single panel or dialog box is active during the grab. However, it is possible for the grab subtree to contain additional toplevel windows; when this happens, all of the panels or dialogs corresponding to those toplevel windows are active during the grab.

Note

Some window managers cause `grab` to fail or to raise an error if you try to set a grab on a window that is not visible. Section 27.3.4 describes how to use the `tkwait visibility` command to wait for a window to become visible.

27.3.2 Local versus Global Grabs

Tk provides two forms of grab, local and global. A *local grab* affects only the grabbing application: if the user moves the pointer into some other application on the display, he or she can interact with the other application as usual. You should always try to use local grabs, and they are the default in the `grab set` command. A *global grab* takes over the entire display so that you cannot interact with any application except the one that set the grab. To request a global grab, specify the `-global` switch to `grab set` as in the following command:

```
grab set -global .dlg
```

Global grabs are rarely needed, and they are tricky to use (if you forget to release the grab, your display locks up so that it is unusable). One place where global grabs are used internally by Tk is for pull-down menus.

Note

If you need to implement a global grab in your application, it can be useful to install some sort of back door to regain control or let you kill the application if you accidentally lock your display. For example, you could install a key binding on the `all` bindtag, or set an `after` event handler to exit your application after some time.

27.3.3 Keyboard Handling during Grabs

Local grabs have no effect on the way the keyboard is handled: keystrokes received anywhere in the application are forwarded to the focus window as usual. Most likely you should set the focus to a window in the grab subtree when you set the grab. Windows outside the grab subtree can't receive any mouse events, so they are unlikely to claim the focus away from the grab subtree. Thus, the grab is likely to have the effect of restricting the keyboard focus to the grab subtree; however, you are free to move the focus anywhere you wish. If you move the pointer into another application, the focus moves to that other application just as if there had been no grab.
During global grabs Tk also sets a grab on the keyboard so that keyboard events go to the grabbing application even if the pointer is over some other application. This means that you cannot use the keyboard to interact with any other application. Once keyboard events arrive at the grabbing application, they are forwarded to the focus window in the usual fashion.

27.3.4 Waiting: The `tkwait` Command

The second aspect of a modal interaction is waiting. Typically you want to suspend a script during a modal interaction and resume it when the interaction is complete. For example, if you display a file selection dialog during a file write operation, you probably want to wait for the user to respond to the dialog, then complete the file write using the name supplied in the dialog interaction. Or when you start up an application, you might wish to display an introductory panel that describes the application and keep this panel visible while the application initializes itself; before going off to do the main initialization, you'll want to be sure that the panel is visible on the screen. The `tkwait` command can be used to wait in situations like these.
`tkwait` has three forms, each of which waits for a different event to occur.

The first form waits for a window to be destroyed, as in the following command:

```
tkwait window .dlg
```

This command does not return until `.dlg` has been destroyed. You might invoke this command after creating a dialog box and setting a grab on it; the command doesn't return until after the user has interacted with the dialog in a way that causes it to be destroyed. While `tkwait` is waiting, the application responds to events so the user can interact with the application's windows. In the dialog box example, bindings must exist to destroy the dialog once the user's response is complete (e.g., the user clicks on the OK button). The bindings for the dialog box might also save additional information in variables (such as the name of a file or an identifier for the button that was pressed). This information can be used once `tkwait` returns.
The following script creates a panel with two buttons labeled "OK" and "Cancel," waits for the user to click on one of the buttons, and then deletes the panel:

```
toplevel .panel
button .panel.ok -text OK -command {
    set label OK
    destroy .panel
}
button .panel.cancel -text Cancel -command {
    set label Cancel
    destroy .panel
}
pack .panel.ok -side left
pack .panel.cancel -side right
grab set .panel
tkwait window .panel
puts "You clicked $label"
```

When the `tkwait` command returns, the variable `label` contains the label of the button on which you clicked.
The second form for `tkwait` waits for the visibility state of a window to change. For example, the command

```
tkwait visibility .intro
```

does not return until the visibility state of `.intro` has changed. Typically this command is invoked just after a new window is created, in which case it doesn't return until the window becomes visible on the screen. `tkwait`

`visibility` can be used to wait for a window to become visible before setting a grab on it, or to make sure that an introductory panel is on the screen before invoking a lengthy initialization script. Like all forms of `tkwait`, `tkwait visibility` responds to events while waiting.
The third form of `tkwait` is equivalent to the `vwait` command. In this form, the command doesn't return until a given variable has been modified. For example, the command

```
tkwait variable x
```

does not return until variable `x` has been modified. This form of `tkwait` is typically used in conjunction with event bindings that modify the variable. For example, the following procedure uses `tkwait variable` to implement something analogous to `tkwait window`, except that you can specify more than one window and the procedure will return as soon as any of the named windows has been deleted (it returns the name of the window that was deleted):

```
proc waitWindows args {
    global dead
    foreach w $args {
        bind $w <Destroy> "set dead $w"
    }
    tkwait variable dead
    return $dead
}
```

27.4 Custom Dialogs

Tk includes a number of built-in dialogs, such as `tk_getOpenFile`, that you should use if they meet your needs. See Section 18.14 for a description of the built-in dialogs provided by Tk. If none of these is suitable, you can create a custom dialog. Your custom dialog should be a toplevel window. You should also register your toplevel window as a transient for your application's main window using a command like the following, as described in Chapter 26:

```
wm transient .dialog .
```

This command registers the toplevel window `.dialog` as a transient window on behalf of the main window ., so that the window manager can choose to provide dialog-specific decorations for your dialog window.

Note

Some window managers create the window as withdrawn if the parent window is withdrawn or iconified. Combined with the grab that you put on the dialog window, this can hang the entire application. Therefore, the safest approach is to make the dialog transient only if the parent is viewable. You could accomplish this with the `winfo viewable` command as follows:

```
if {[winfo viewable .]} {
    wm transient .dialog .
}
```

When you have a lot of widgets for your dialog, you can hide the window until all the widgets are ready. In window manager terms, you withdraw the window and then later deiconify it (all these terms come from the X Window System); for example:

```
toplevel .dialog
wm withdraw .dialog

# Create and manage all the widgets...

wm deiconify .dialog
```

You should also set the title and a script to handle the close button (as described in Chapter 26); for example:

```
wm title .dialog "Create User Account"
wm protocol .dialog WM_DELETE_WINDOW "cancelDialog"
```

This example calls the procedure `cancelDialog` when the user clicks on the close button for the window. If your dialog has a “Cancel” button, it’s often a good idea to use the same script as for that button. In fact, you can simply invoke the “Cancel” button’s script in this case, for example:

```
wm protocol .dialog WM_DELETE_WINDOW {
    .dialog.cancel invoke
}
```

For data-entry widgets, you normally use a grid-based layout to manage the widget positions. Using a grid allows the prompts and data-entry widgets to line up. And if you want the dialog to be modal, follow the techniques described in this chapter; for example:

```
wm deiconify .dialog
tkwait visibility .dialog
grab set .dialog
tkwait window .dialog
```

These commands wait until the dialog window is destroyed.

Note

On some window managers, it is possible that these `tkwait` or `grab set` commands will raise an error condition. Therefore, it's safest to run each of these using a `catch` command to silently ignore these errors.

The following script pulls all these pieces together to create a simple main window and a modal dialog. In this case, the dialog gathers input to set up a user account for some hypothetical Internet site. To keep the example simple, the main window sports two buttons, one to display the "Create Account" dialog and one to exit the application. The script follows:

```
# Tk custom dialog example

proc cancelDialog {} {
    destroy .dialog
}

proc createAccount {} {
    global username password firstname lastname
    puts "Create account for $firstname $lastname."
    puts "With user name: $username password $password."

    destroy .dialog
}

proc showCreateAccount {} {
    global username password firstname lastname
    set firstname ""
    set lastname ""
    set username ""
    set password ""


    toplevel .dialog
    wm withdraw .dialog

    ttk::frame .dialog.f -relief flat

    # Login
    ttk::labelframe .dialog.f.login -text "Login:"
    ttk::label .dialog.f.login.userl -text "User Name:"
    ttk::entry .dialog.f.login.usert -textvariable username
    ttk::label .dialog.f.login.passl -text "Password:"
    ttk::entry .dialog.f.login.passt -show "*" \
            -textvariable password
```

```
grid config .dialog.f.login.userl \
    -column 0 -row 0 -sticky e
grid config .dialog.f.login.passl \
    -column 0 -row 1 -sticky e
grid config .dialog.f.login.usert \
    -column 1 -row 0 -sticky snew
grid config .dialog.f.login.passt \
    -column 1 -row 1 -sticky snew

pack .dialog.f.login -padx 5 -pady 10


# User information
ttk::labelframe .dialog.f.userinfo \
    -text "User Information:"
ttk::label .dialog.f.userinfo.firstl -text "First Name:"
ttk::entry .dialog.f.userinfo.firstt \
    -textvariable firstname
ttk::label .dialog.f.userinfo.lastl -text "Last Name:"
ttk::entry .dialog.f.userinfo.lastt \
    -textvariable lastname



grid config .dialog.f.userinfo.firstl \
    -column 0 -row 0 -sticky e
grid config .dialog.f.userinfo.lastl \
    -column 0 -row 1 -sticky e
grid config .dialog.f.userinfo.firstt \
    -column 1 -row 0 -sticky snew
grid config .dialog.f.userinfo.lastt \
    -column 1 -row 1 -sticky snew

pack .dialog.f.userinfo -padx 5 -pady 10


# Action buttons
ttk::frame .dialog.f.buttons -relief flat

ttk::button .dialog.f.buttons.ok -text "Create Account" \
    -command {createAccount}
ttk::button .dialog.f.buttons.cancel -text "Cancel" \
    -command {cancelDialog}

pack .dialog.f.buttons.ok -side left
pack .dialog.f.buttons.cancel -side right
pack .dialog.f.buttons -padx 5 -pady 10
pack .dialog.f
```

```
        # Window manager settings for dialog
        wm title .dialog "Create User Account"
        wm protocol .dialog WM_DELETE_WINDOW {
            .dialog.f.buttons.cancel invoke
        }
        wm transient .dialog .

        # Ready to display the dialog
        wm deiconify .dialog

        # Make this a modal dialog.
        catch {tk visibility .dialog}
        focus .dialog.f.login.usert
        catch {grab set .dialog}
        catch {tkwait window .dialog}
    }

    ttk::button .createAccount -text "Create Account..." \
       -command {showCreateAccount}

    ttk::button .exitButton -text "Exit" -command { exit }

    pack .createAccount .exitButton -padx 10 -pady 20
```

In this script, the `cancelDialog` procedure destroys the dialog window. The `createAccount` procedure acts as a placeholder for the actual code that would create a new user account. Figure 27.1 shows the main window. The `showCreateAccount` procedure creates and populates the create account dialog when the user clicks on the “Create Account” button. Figure 27.2 shows this dialog. While the “Create Account” dialog is visible, the main window does not accept keyboard or mouse input. The user then can enter in account information, including a password that is shown as asterisks.

Figure 27.1 The main window

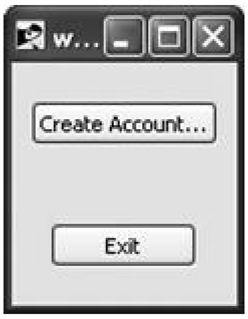
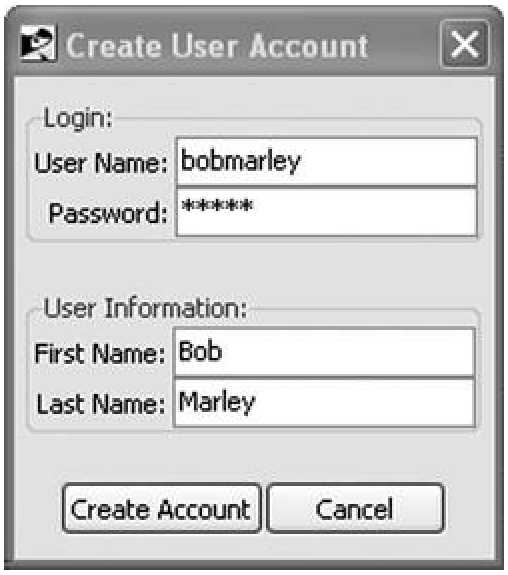


Figure 27.2 The create account custom dialog


Create User Account

Login:

User Name: bobmarley

Password: *****

User Information:

First Name: Bob

Last Name: Marley

Note

An alternate approach would be to build the dialog once, and then use `wm deiconify` and `wm withdraw` to show and hide it as needed. This could save time if it were a particularly complex dialog. It also has the benefit of allowing widgets to maintain—and even update—their state while hidden, which could be quite useful for something like a preferences dialog.

28. More on Configuration Options

Configuration options for widgets were introduced in previous chapters. You can specify configuration options when creating new widgets and modify them with the `configure` action for widget commands. This chapter describes two additional facilities related to options. The first part of the chapter describes the *option database*, which can be used to specify default values for options. The second part of the chapter describes the full syntax of the `configure` widget command; it can be used to retrieve information about options as well as to modify options.

28.1 Commands Presented in This Chapter

This chapter discusses the following commands for manipulating configuration options. For a complete list of the options available for a particular widget class, see the reference documentation for the corresponding class command, such as `button`.

- `class widget ?optionName value optionName value ...?`

Creates a new widget with class `class` and path name `widget` and sets options for the new widget as given by `optionName value` pairs. Unspecified options are filled in using the option database or widget defaults. Returns `widget` as the result.

- `widget cget optionName`

Returns the value currently assigned to the option `optionName` for `widget`.

- `widget configure`

Returns a list whose elements are sublists describing all of the options for `widget`. Each sublist describes one option in the form described next.

- `widget configure optionName`

Returns a list describing the option `optionName` for `widget`. The list normally contains five values: `optionName`, the option's name in the option database, its class, its default value, and its current value. If the option is a synonym for another option, the list contains two values: the option name and the database name for the synonym.

- `widget configure optionName value ?optionName value ...?`

Sets the value for each `optionName` of `widget` to the corresponding `value`.

- `option add pattern value ?priority?`

Adds a new option to the option database as specified by *pattern* and *value*. *priority* must be either a number between 0 and 100 or a symbolic name (see the reference documentation for details on symbolic names); if omitted, it defaults to `interactive` (`80`).

- `option clear`

Removes all entries from the option database.

- `option get` *widget dbName dbClass*

If the option database contains a pattern that matches *widget*, *dbName*, and *dbClass*, returns the value for the highest-priority matching pattern. Otherwise returns an empty string.

- `option readfile` *fileName* ?*priority*?

Reads *fileName*, which must have the standard format for a `.Xdefaults` file, and adds all the options specified in that file to the option database at priority level *priority*. The priority defaults to `interactive` (`80`) if omitted.

28.2 The Option Database

The option database supplies values for configuration options that aren't specified explicitly by the application designer. The option database is consulted when widgets are created: for each option not specified on the command line, the widget queries the option database and uses the value from the database, if there is one. If there is no value in the option database, the widget class supplies a default value. Values in the option database are usually provided by the user to personalize applications, for example, to specify consistently larger fonts. On Unix systems, Tk supports the RESOURCE_MANAGER property and `.Xdefaults` file in the same way as other X toolkits.

The option database shouldn't be needed very often in Tk applications because widgets have reasonable default values for their options. If options do need to be changed, it is often easier to make the changes by invoking `configure` widget commands from a Tcl script rather than creating entries in your `.Xdefaults` file. The option database exists primarily to provide cultural compatibility with other X toolkits; you should use it as little as possible.

28.3 Option Database Entries

The option database contains any number of entries, where each entry

consists of two strings: a *pattern* and a *value*. The pattern specifies one or more widgets and options, and the value is a string to use for options that match the pattern.
In its simplest form a pattern consists of an application name, an optional widget name, and an option name, all separated by dots. For example, here are two patterns in this form:

```
wish.a.b.foreground
wish.background
```

The first pattern applies to the `foreground` option in the widget `.a.b` in the application `wish`. In the second pattern the widget name is omitted, so the pattern applies to the main widget for `wish`. Each of these patterns applies to only a single option for a single widget.
Patterns may also contain classes or wildcards, which allow them to match many different options or widgets. Any component of the widget name may be replaced by a class, in which case the pattern matches any widget that is an instance of that class. For example, the following pattern applies to all children of `.a` that are checkbuttons:

```
wish.a.Checkbutton.foreground
```

Application and option names may also be replaced with classes. The application name is the name of the executable (which would be the interpreter name if you start an interactive interpreter, or the script name if you create a self-executing script); the application class is the application name with the initial letter capitalized. Individual options also have classes. For example, the class for the `foreground` option is `Foreground`. Several other options such as `insertBackground` (the color used for displaying an insertion cursor) also have the class `Foreground`, so the following pattern applies to any of these options for any entry widget that is a child of `.a` in `wish`:

```
wish.a.Entry.Foreground
```

Last, patterns may contain `*` wildcard characters. An `*` matches any number of window names or classes, as in the following examples:

```
*Foreground
*Button.foreground
```

The first pattern applies to any option in any widget of any application as long as the option's class is `Foreground`. The second pattern applies to the `foreground` option of any button widget in any application. The `*` wildcard may be used only for window or application names; it cannot be used for the option name (it wouldn't make much sense to specify the same value for all options of a widget).

Note

This syntax for patterns is the same as that supported by the standard X resource database mechanisms in the X Window System.

The database name for an option is usually the same as the name you would use in a widget creation command or a `configure` widget command, except that there is no leading `-` and capital letters are used to mark internal word boundaries. For example, the database name for the `-borderwidth` option is `borderWidth`. The class for an option is usually the same as its database name except that the first letter is capitalized. For example, the class for the `-borderwidth` option is `BorderWidth`. It's important to remember that in Tk classes *always* start with an initial capital letter; any name starting with an initial capital letter is assumed to be a class.

28.4 The `RESOURCE_MANAGER` Property and `.Xdefaults` File

When a Tk application starts up, Tk automatically initializes the option database. For an X windowing system, if there is a `RESOURCE_MANAGER` property on the root window for the display, the database is initialized from it. Otherwise, Tk checks the user's home directory for a `.Xdefaults` file and uses it if it exists. The initialization information has the same form whether it comes from the `RESOURCE_MANAGER` property or the `.Xdefaults` file. The syntax described here is the same as that supported by other X toolkits.
Each line of initialization data specifies one entry in the resource database in a form such as the following:

```
*Foreground: blue
```

The line consists of a pattern (`*Foreground` in the example) followed by a colon, whitespace, and a value to associate with the pattern (`blue` in the example). If the value is too long to fit on one line, it can be placed on multiple lines with each line but the last ending in a backslash-newline sequence:

```
*Gizmo.text: This is a very long initial \
value to use for the text option in all \
"Gizmo" widgets.
```

The backslashes and newlines are not part of the value.
Blank lines are ignored, as are lines whose first nonblank character is `#` or `!`.

28.5 Priorities in the Option Database

It is possible for several patterns in the option database to match a particular option. When this happens, Tk uses a two-part priority scheme to determine which pattern applies. Tk's mechanism for resolving conflicts is different from the standard mechanism supported by the X Toolkit (Xt).
For the most part the priority of an option in the database is determined by the order in which it was entered into the database: newer options take priority over older ones. When specifying options (for example, by typing them into your `.Xdefaults` file), you should specify the more general options first, and more specific overrides later. For example, if you want button widgets to have a background color of `Bisque1` and all other widgets to have white backgrounds, put the following lines in your `.Xdefaults` file:

```
*background: white
*Button.background: Bisque1
```

The `*background` pattern matches any option that the `*Button.background` pattern matches, but the `*Button.background` pattern has higher priority since it was specified last. If the order of the patterns had been reversed, all widgets (including buttons) would have white backgrounds, and the `*Button.background` pattern would have no effect.
In some cases it may not be possible to specify general patterns before specific ones (for example, you might add a more general pattern to the option database after it has already been initialized with a number of specific patterns from the `RESOURCE_MANAGER` property). To accommodate these situations, each entry also has an integer priority level between 0 and 100, inclusive. An entry with a higher-priority level takes precedence over entries with lower-priority levels, regardless of the order in which they were inserted into the option database. Priority levels are not used very often in Tk; refer to the reference documentation for complete details on

how they work.
Tk’s priority scheme is different from the scheme used by other X toolkits such as Xt. Xt gives higher priority to the most specific pattern; for example, `.a.b.foreground` is more specific than `*foreground`, so it receives higher priority regardless of the order in which the patterns appear. In most cases this isn’t a problem: you can specify options for Xt applications using the Xt rules, and options for Tk applications using the Tk rules. In cases where you want to specify options that apply to both Tk applications and Xt applications, use the Xt rules but make sure that the patterns considered higher-priority by Xt appear later in your `.Xdefaults` file. In general, you shouldn’t need to specify very many options to Tk applications (the defaults should always be reasonable), so the issue of pattern priority shouldn’t come up often.

Note

The option database is queried only for options not specified explicitly in the widget creation command. This means that the user cannot override any option that is specified in a widget creation command. If you want to specify a value for an option but allow the user to override that value through the `RESOURCE_MANAGER` property, you should specify the value for the option using the `option` command, described in the next section.

28.6 The `option` Command

The `option` command allows you to manipulate the option database while an application is running. The command `option add` creates a new entry in the database and takes two or three arguments. The first two arguments are the pattern and value for the new entry, and the third argument, if specified, is a priority level for the new entry. The priority defaults to `interactive` (`80`) if omitted. For example,

```
option add *Button.background Bisque1
```

adds an entry that sets the background color for all button widgets to `Bisque1`. Changes to the option database affect only the application in which `option add` is invoked, and they apply only to new widgets created after `option add` is

executed; the database changes do not affect widgets that already exist.
The `option clear` command removes all entries from the option database. On the next access to the database, it is re-initialized from the `RESOURCE_MANAGER` property or the `.Xdefaults` file.
The command `option readfile` reads a file in the format described earlier for the `RESOURCE_MANAGER` property and make entries in the option database for each line. For example, the following command augments the option database with the information in the file `newOptions`:

```
option readfile newOptions
```

The `option readfile` command can also be given a priority level as an extra argument after the file name to specify the priority at which the options are added. The priority defaults to `interactive` (`80`) if omitted.
To query whether there is an entry in the option database that applies to a particular option, use the `option get` command:

```
option get .a.b background Background
```

This command takes three arguments: the path name of a widget (`.a.b`), the database name for an option (`background`), and the class for that option (`Background`). The command searches the option database to see if any entries match the given window, option, and class. If so, the value of the highest-priority matching option is returned. If no entry matches, an empty string is returned.

28.7 The `configure` Widget Command

Every widget class supports a `configure` widget command. This command comes in three forms, which can be used both to change the values of options and to retrieve information about the widget's options.
If a `configure` widget command is given two additional arguments, it changes the value of an option, as in the following example:

```
.button configure -text Quit
```

If the `configure` widget command is given just one extra argument, it returns information about the named option:

```
   .button configure -text
⇒ -text text Text { } Quit
```

The return value is normally a list with five elements. The first element of the list is the name of the option as you'd specify it on a Tcl command line when creating or configuring a widget. The second and third elements are a name and class to use for looking up the option in the option database. The fourth element is the default value provided by the widget class (a single space character in the preceding example), and the fifth element is the current value of the option.
Some widget options are just synonyms for other options (e.g., the `-bg` option for buttons is the same as the `-background` option). Configuration information for a synonym is returned as a list with two elements consisting of the option's command-line name and the option database name of its synonym:

```
  .button configure -bg
⇒ -bg background
```

If the `configure` widget command is invoked with no additional arguments, it returns information about all of the widget's options as a list of lists, where each element is a nested sublist of information for each option:

```
  .button configure
⇒ {-activebackground activeBackground Foreground
  systemButtonFacePressed systemButtonFacePressed}
  {-activeforeground activeForeground Background
  systemPushButtonPressedText systemPushButtonPressedText} {-anchor
  anchor Anchor center center} {-background background Background
  White White} {-bd -borderwidth} {-bg -background} {-bitmap bitmap
  Bitmap {} {}} {-borderwidth borderWidth BorderWidth 2 2} {-command
  command Command {} {}} {-compound compound Compound none none}
  {-cursor cursor Cursor {} {}} {-default default Default disabled
  disabled} {-disabledforeground disabledForeground
  DisabledForeground #a3a3a3 #a3a3a3} {-fg -foreground} {-font font
  Font TkDefaultFont TkDefaultFont} {-foreground foreground
  Foreground systemButtonText systemButtonText} {-height height
  Height 0 0} {-highlightbackground highlightBackground
  HighlightBackground White White} {-highlightcolor highlightColor
  HighlightColor systemButtonFrame systemButtonFrame}
  {-highlightthickness highlightThickness HighlightThickness 4 4}
  {-image image Image {} {}} {-justify justify Justify center
  center} {-overrelief overRelief OverRelief {} {}} {-padx padX Pad
  12 12} {-pady padY Pad 3 3} {-relief relief Relief flat flat}
  {-repeatdelay repeatDelay RepeatDelay 0 0} {-repeatinterval
  repeatInterval RepeatInterval 0 0} {-state state State normal
  normal} {-takefocus takeFocus TakeFocus {} {}} {-text text Text {}
  Quit} {-textvariable textVariable Variable {} {}} {-underline
  underline Underline -1 -1} {-width width Width 0 0} {-wraplength
  wrapLength WrapLength 0 0}
```

28.8 The `cget` Widget Command

Every widget class supports a `cget` widget command. This command accepts the name of one option as an argument, and it returns the current value of that option. Pass the name of the option. For example:

```
  .button cget -text
⇒ Quit
  .button cget -background
⇒ White
  .button cget -padx
⇒ 12
  .button configure -padx 150
  .button cget -padx
⇒ 150
```

Unlike the `configure` widget command, which returns a list when used to retrieve widget values, the `cget` widget command returns just the value of the given option.

29. Odds and Ends

This chapter describes several additional Tk commands: `destroy`, which deletes widgets; `update`, which forces operations that are normally delayed, such as screen updates, to be done immediately; `winfo`, which provides a variety of information about windows, such as their dimensions and children; `bell`, for ringing the bell; and `tk`, which provides access to various internals of the Tk toolkit. This chapter also describes several predefined variables that are read or written by Tk and may be useful in Tk applications.

29.1 Commands Presented in This Chapter

This chapter discusses the following commands:

- `destroy` *`window`* `?`*`window window`* `...?`

Deletes each of the *`window`*s and all of the windows descended from them. The corresponding widget commands (and all widget states) are also deleted.

- `tk appname ?`*`newName`*`?`

Returns the current application name, or sets the application name to *`newName`*.

- `tk inactive ?-displayof` *`window`*`? ?reset?`

Returns the time in milliseconds since the last user interaction on the display containing *`window`*, which defaults to .. Including the `reset` argument resets the idle timer.

- `tk scaling ?-displayof` *`window?`* `?`*`number`*`?`

Returns the scaling factor on the display containing *`window`*, which defaults to ., or sets it to *`number`*. The scaling factor is a floating-point number expressing the number of pixels per typographic point.

- `tk windowingsystem`

Returns the current Tk windowing system, one of `x11` (X11-based), `win32` (MS Windows), or `aqua` (Mac OS X Aqua).

- `update ?idletasks?`

Brings the display up to date and processes all pending events. If `idletasks` is specified, no events are processed except those in the idle task queue (delayed updates).

- `winfo` *`option`* `?`*`arg arg`* `...?`

Returns various pieces of information about windows, depending on the `option` argument. See the reference documentation for details.

29.2 Destroying Widgets

The `destroy` command deletes one or more widgets. It takes any number of widget names as arguments; for example:

```
destroy .dlg1 .dlg2
```

This command destroys `.dlg1` and `.dlg2`, including their widget state and the widget commands named after the windows. It also recursively destroys their children. The command `destroy .` destroys all of the widgets in the application; when this happens, most Tk applications exit.

29.3 The `update` Command

Tk normally delays operations such as screen updates until the application is idle. For example, if you invoke a widget command to change the text in a button, the button doesn't redisplay itself immediately. Instead, it schedules the redisplay to be done later and returns immediately. At some point the application becomes *idle*, which means that all existing events have been processed and the application is in the event loop and about to wait for another event to occur. At this point all of the delayed operations are carried out. Tk delays redisplays because it saves work when the same window is modified repeatedly: with delayed redisplay the window gets redrawn only once at the end. Tk also delays many other operations, such as geometry recalculations and window creation.
For the most part the delays are invisible. Interactive applications rarely do very much work at a time, so Tk becomes idle again very quickly and updates the screen before the user can perceive any delay. However, there are times when the delays are inconvenient. For example, if a script is going to execute for a long time, you may wish to bring the screen up to date at certain times during its execution. The `update` command allows you to do this. If you invoke the command

```
update idletasks
```

all of the delayed operations such as redisplays are carried out immediately; the command does not return until they have finished.

Note

Like the `after` command, the `update` command used to be part of Tk. It has since moved to the Tcl core language, so you can use the `update` command in applications such as network servers that don't need to present a graphical user interface.

The following procedure uses `update` to flash a widget synchronously:

```
proc flash {w option value1 value2 interval count} {
    for {set i 0} {$i < $count} {incr i} {
        $w config $option $value1
        update idletasks
        after $interval
        $w config $option $value2
        update idletasks
        after $interval
    }
}
```

This procedure flashes the widget a given number of times and doesn't return until the flashing is complete. Tk never becomes idle during the execution of this procedure (the `after` command doesn't return to the event loop while it waits for the time to elapse), so the `update` commands are needed to force the widget to be redisplayed. Without the `update` commands no changes would appear on the screen until the script completed, at which point the widget's option would change to `value2`.
If you invoke `update` without the `idletasks` argument, all pending events are processed, too. You might do this in the middle of a long calculation to allow the application to respond to user interactions (for example, the user might invoke a *Cancel* button to abort the calculation).

Note

The `update` command can be dangerous, in that it starts an instance of Tcl's event loop. If your script executes an `update` command from within an event handler script, it actually starts a nested instance of an event loop that must terminate before the outer event loop regains control. This can cause serious problems with your control flow, especially if the outer loop was started by a `vwait` or `tkwait` command. If the event that they were looking for occurs during the `nested` update event loop, they don't "see" the event and so don't terminate as expected. You should avoid nested event loops wherever possible. One approach is to break a long-running activity into smaller chunks and have each chunk schedule the next chunk to execute via the `after` command. Another approach is to use the `Thread` extension to create a multithreaded Tcl application, so that a long-running activity in one thread doesn't starve the event loop in another thread.

29.4 Information about Widgets

The `winfo` command provides information about widgets. It has almost 50 different subcommands for retrieving different kinds of information about a widget. For example, `winfo exists` returns a `0` or `1` value to indicate whether a widget exists. `winfo children` returns a list whose elements are the children of the widget. `winfo width` and `winfo height` return the current dimensions of the widget. `winfo class` returns the class of the widget, such as `Button` or `Text`. Refer to the Tk reference documentation for details on all of the options provided by `winfo`.

29.5 The `tk` Command

The `tk` command provides access to various aspects of Tk's internal state. Most everything available from this command pertains to Tk in its entirety, the application, or the overall display.
It can be important to know the windowing system on which your application is executing. Historically, people tested the value of the `::tcl_platform(platform)` array element, which Tcl would set to one of `windows`, `unix`, or `macintosh`. However, since the introduction of Mac OS X, this array element now (correctly) reports `unix` for a Mac OS X system. The `tk windowingsystem` command is now the correct way to determine the windowing system. It returns a value of `aqua`, `win32`, or `x11`.
`tk inactive` returns the number of milliseconds since a user last interacted

with the system, by moving the mouse, pressing a key, and so on:

```
   tk inactive
⇒ 3
```

`tk inactive` returns `-1` if executed in a safe interpreter or on a system that doesn't support querying the inactive time. You can request the idle time of a particular display by including the `-displayof` option and specifying the name of a window on that display. Additionally, you can include the argument `reset` to reset the idle timer.
`tk scaling` returns the scaling factor used when converting between pixels and physical units of measure. It's expressed as a floating-point number, giving the number of pixels per typographic point (1/72 inch), and you can use the `-displayof` option to specify the name of a window on a particular display whose scaling factor you want to retrieve:

```
   tk scaling
⇒ 1.000492368291482
```

Tk does its best to determine the appropriate value when the application starts, but systems often don't provide Tk with a truly accurate value. If an accurate scaling factor is important to your application, you should give your users some method for calibrating the on-screen measurement. You can then pass the new scaling factor to `tk scaling` as an argument to set the scaling factor of the display for your application. The following example uses the `winfo screenmmwidth` command to retrieve the calculated width in millimeters of a window on-screen:

```
   winfo screenmmwidth .
⇒ 677
   tk scaling 1.25
   winfo screenmmwidth .
⇒ 542
```

If your application uses the Tk `send` command to send messages to other Tk applications, you need to know, and sometimes set, the name of the application. The `tk appname` command returns the current name of the application or sets it to a new value. See Chapter 12 for more information on the `send` command.

29.6 Variables Managed by Tk

Several global variables are significant to Tk, either because Tk sets them or because it reads them and adjusts its behavior accordingly. You may find the following variables useful:

- `tk_library`

Set by Tk to hold the path name of the directory containing a library of standard Tk scripts and demonstrations. This variable is set from the `TK_LIBRARY` environment variable, if it exists, or from a set of other standard locations otherwise. See the `tkvars` reference documentation for a detailed description.

- `tk_version`

Set by Tk to its current version number. It has a form like `8.5`, where `8` is the major version number and `5` is a minor version number. Changes in the major version number imply incompatible changes in Tk.

- `tk_patchLevel`

Set by Tk to its current patch level. It has a form like `8.5.4`, where `8` is the major version number, `5` is a minor version number, and `4` is the specific patch level.

In addition to these variables, which may be useful to the application, Tk also uses the associative array `tk::Priv` to store information for its private use. Applications should not normally use or modify any of the values in `tk::Priv`.

29.7 Ringing the Bell

Use the `bell` command to ring the bell, or produce a visual effect, on a given display. The basic format is

```
bell
```

You can also pass the `-displayof` option to define which display should have its bell rung:

bell -displayof *window*

With the `-nice` option, the `bell` command attempts to reset the screen saver on the given display, normally making the screen visible again.

30. Tcl and C Integration Philosophy

While it is possible to write a large variety of applications completely in Tcl, in some cases it is useful or necessary to combine Tcl and C code, and it is common to find large applications with portions written in both.

Where possible, it is almost always best to write Tcl code instead of using the C programming language. Tcl programs are easier to write and quicker to modify, require less expertise, and are therefore more accessible to other programmers. Scripts do not require recompilation after every change and are also generally less difficult to debug. However, there are times when programming in C is the best choice.

Fortunately, Tcl makes it very pleasant and productive to combine Tcl and C, having been designed with this in mind from the outset.

The two most common reasons for mixing Tcl and C are

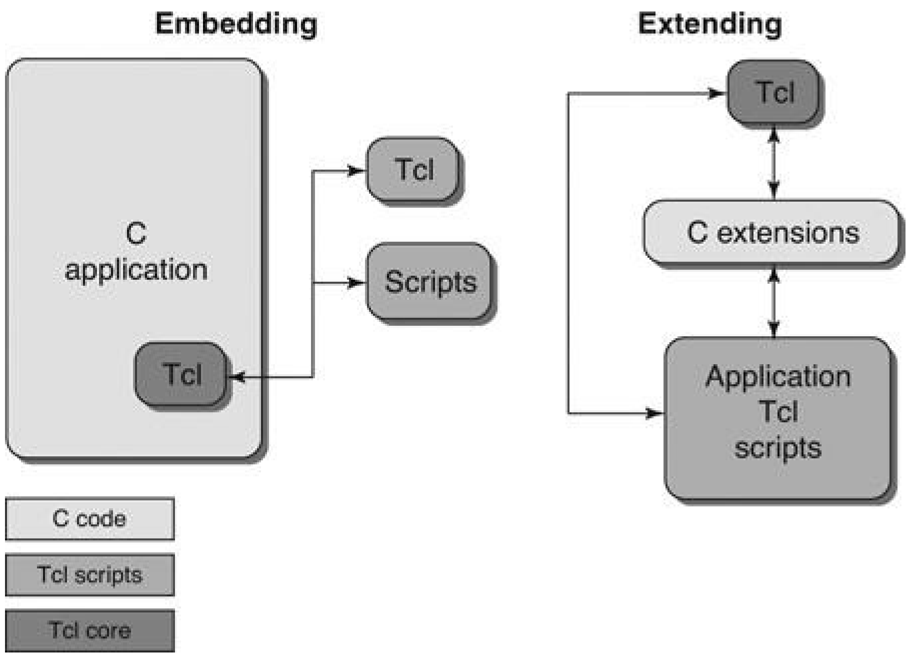
- Adding functionality not available to Tcl, for example, creating a Tcl interface to an image manipulation library written in C, or writing code to talk to a special device on a Linux system that is accessed via specific arguments to the `ioctl` system call.
- Optimizing performance-intensive tasks that must run as fast as possible. C is many times faster than Tcl for processor-intensive operations, and is therefore the better choice where speed is critical. Tasks involving extensive numerical computation or manipulation of binary-formatted information often benefit from implementation in C code.

This part of the book deals with how to get the best of both worlds by using the two languages together, each being used for the part of the application for which it is best suited. You will find that it's easy and a very powerful way of programming that can lead to new and innovative applications.

Programmers generally add C code to Tcl applications by creating their own Tcl commands, written in C. These commands then can be bundled into extensions, which are loaded into the application as needed. These new commands act similarly to any of the built-in Tcl commands or procedures that you've created, making them easy to control from Tcl. If you are writing C code to improve your program's performance, an intelligent approach to optimization is to begin with a program written entirely in Tcl, analyze it for slow spots, and then rewrite those spots in C. This is a very attractive style of developing an application, because it confers on the programmer all the advantages of using a high-level language and concentrates low-level C work only where it is absolutely necessary, maximizing both programmer time and program efficiency.

A second approach is to embed Tcl in an existing application, a powerful technique to enhance your application's existing capabilities with a scripting language. When done properly, this can turn the program into a dynamic, configurable system whose behavior may be modified at runtime via Tcl scripts, instead of a rigid block of code that must be modified and recompiled in order to change its behavior, or at least restarted to reread configuration files. As an example, with relatively few lines of C code, the Rivet Apache web server module makes it possible to create dynamic web pages that rival PHP for speed and ease of web programming. Figure 30.1 illustrates two methods of integrating C code and Tcl.

Figure 30.1 Embedding versus extending Tcl



There are several parts to most Tcl/C integration projects. The majority of the code implements your special-purpose Tcl commands. Then comes the startup code that registers those commands with the Tcl interpreter and takes care of any other startup tasks such as creating global variables. If you're writing an application that embeds Tcl, at some point you need to write code that evaluates Tcl scripts. On the other hand, if you're writing an extension, your C code probably won't need to evaluate scripts; having the extension register its commands with the Tcl interpreter usually is sufficient.

30.1 Tcl versus C: Where to Draw the Line

In order to maximize the flexibility of your application, it is best to organize your C code as a group of primitive operations—basic building blocks, flexible enough to be used in different ways—instead of attempting to make every possible option available as C code. This makes it possible to mix and match these operations in your Tcl scripts. The most important decision to make is where to draw the line between having one big, simple command that does it all and having many small, very fine-grained commands. If you hide too much functionality behind one command, it may not be possible to write scripts to combine the functionality in new and interesting ways that the original author hadn't planned for. On the other hand, creating commands that are too low-level may not really provide any benefits or added flexibility to developers using them. The right point to separate Tcl from C is where you have maximum flexibility, but before your Tcl commands become a repetitive copy of the C API.

Tcl's `socket` command is a good example of an interface to a complex underlying system. It does not mirror the series of C calls necessary to open a socket. Instead, all you do is give it an address (which may be a name or an IP number) and a port, and it hands you a channel that you can write to or read from. More advanced configuration is still possible via the `fconfigure` command, but the basic operation is very easy.

As another example, consider an array of sensors monitoring some information about a production line in a plant, where each sensor takes environmental measurements at a different point in the process. Users of your interface may wish to perform tasks such as

- Printing a complete report of the current state of the production run to a file on disk
- Saving which station has the lowest energy consumption in a database
- Displaying the temperature at station X
- Returning which stations spend more than 25% of their time idle
- Passing the average, high, and low temperatures to a web server

You'll need to write some C code to access this information, which is normally available only as a low-level API provided by the operating system.

One approach might be to write code that reads all information from all sensors and dumps it to standard output. It's certainly a quick approach, but it's not going to be very flexible in the long term and might also be slow, depending on access times for the sensors.

Another tactic might be to provide Tcl commands to select a sensor, determine which information is available, open it for reading, and then fetch one measurement. This is certainly flexible, but it's going to be slow to code using the resulting API.
One way to strike a good balance would be to provide a `sensor` command that takes several arguments:

- `list`—returns a list of all sensors
- `ready`—returns a list of all sensors currently online and transmitting data
- `read $sensor`—reads and returns the information for a specific sensor

This would make it easy to use standard Tcl commands like `foreach` to loop through the list of sensors, taking readings from each one. Although the `sensor` command doesn't return an immediate answer for any of the tasks listed previously, it's possible to use it, together with a bit of scripting, to quickly develop solutions for all five problems, and yet it is not too low-level.
Printing the entire report to a file would be a matter of looping over all stations, reading them, and then printing them to a file via the `open` and `puts` commands. The lowest energy consumption could be determined by looping over the online sensors and saving progressively lower values as they appear—and at that point the results could be saved to a relational database. Displaying one value of the reading for a particular sensor is as easy as reading it and saving only the desired information, discarding the rest. Collecting statistics on idle stations would, once again, involve reading all stations, then looking at one particular value from the collected results.

30.2 Resource Names—Connecting C Constructs to Tcl

One common usage of C extensions is to allow Tcl code to interact with complex data structures or objects from either your own application or another C library. In your C program, you'd normally refer to this data by a pointer to the data structure or object.
Since everything in Tcl ought to be representable as a string (so we can't use pointers), we need to dedicate some thought to connecting these abstract strings with what they represent at the C level—most likely C structures. In the example in the preceding section, the sensors might be given names relative to their position or function, such as `1-heat`, `2-pressure`, `3-oxygen`. These names are brief but descriptive, so that people have an idea of what the

object might be just by looking at the name. In an example of this system at work, Tcl uses names like `file1` and `sock7` for file channel handles and socket channel handles respectively. These are simple, clear names that give the user an idea that the thing in question is a socket or file. In this case we also utilize the POSIX file descriptor number to create a unique name, although that information isn't meant to be used and shouldn't be relied on at the scripting level. In turn, these strings can map to specific structures, most likely via a hash table that, for every string input, is able to return a C structure of the desired type. Tcl contains a powerful hash table implementation that makes this approach easy to implement. See Chapter 40 for more information on Tcl's hash table functionality.

30.3 "Action-Oriented" versus "Object-Oriented"

There are two approaches to defining commands in an application: "action-oriented" and "object-oriented." In the action-oriented approach, commands are defined that act on objects passed to them as parameters; for instance, Tcl's commands for manipulating files all take a file identifier returned by `open` as an argument and proceed to perform some action on it:

```
set fid [open "/tmp/foobar" w]
fconfigure $fid -translation lf
puts $fid "Some text"
close $fid
```

In contrast, the "object-oriented" style provides one command whose only job is to represent the object directly. In this case, there are many Tcl commands that can be called: one for each object, and one to create new objects. In retrospect, it might have been more logical to adopt this style for files: you would create a file with `open`, and then use it as a command, as here:

```
set fid [file "/tmp/foobar" w]
$fid fconfigure -translation lf
$fid puts "Some text"
$fid close
```

Despite appearances, remember that commands created in this way are just regular Tcl commands. While in many ways they act similarly to objects in languages like Smalltalk, Ruby, or Java (that is, the subcommands look like "methods"; most of them have some data associated with them; etc.), they actually are procedural commands; they do not have inheritance or support other features of true object-oriented programming. There are other ways (several, in fact) to create objects in Tcl based on classes, methods, inheritance, and so on.

The action-oriented style is best when

- There are a large number of resources (e.g., strings or integers), so it would be wasteful to create a command for each one.
- The resources are short-lived, so setup and tear-down time for the object would be more effort than it was worth.
- A command accepts many types of objects as parameters and executes the same code on all of them. For example, the `destroy` command in Tk works on all widget types, so it's best to keep all the functionality in one place, instead of giving every widget a `destroy` subcommand.

Our sensor-monitoring example uses this approach because there may be a large number of sensors to watch, and there are not many actions that may be performed on each one. In the case that not all sensors are read constantly, it makes even more sense to use the action-oriented approach, so that the program does not have to maintain information for each one all the time but only when requested.

Using the "object-oriented" approach works well when

- There are not too many objects (tens or hundreds).
- The objects are well defined.
- They are likely to be in existence and used for a reasonable duration.
- They are reasonably complex, with a number of operations that may be performed on them.

30.4 Representing Information

The information passed into and out of your Tcl commands should be formatted for easy processing by Tcl scripts, not necessarily for maximum human readability. The sensor monitor commands should not return a description of the sensor status like `hot`, `real hot`, or `The temperature is 70 C`, or a even nicely formatted table ready for printing, but rather results easily passed to and consumed by other Tcl commands. Should you need to structure the data in some way, you can create Tcl lists and arrays (or dictionaries in Tcl 8.5) from C. As an aid to other programmers or consumers of your data, it is preferable to make your data self-descriptive enough that you don't leave people wondering about what position in a list corresponds to what value. For instance, the list `105 89 96` tells us nothing by itself. If we use a dictionary like `max 105 min 89 average 96`, the data makes much more sense at a glance, and yet it is still very easy to manipulate from Tcl.

31. Interpreters

This chapter explains what interpreters are, how to create them and delete them, and how to use them to evaluate Tcl scripts.

31.1 Functions Presented in This Chapter

- `Tcl_Interp *Tcl_CreateInterp()`

Creates and returns a new Tcl interpreter.

- `Tcl_DeleteInterp(Tcl_Interp *interp)`

Deletes a Tcl interpreter.

- `Tcl_InterpDeleted(Tcl_Interp *interp)`

Returns nonzero if the interpreter is slated for deletion.

- `Tcl_Interp *Tcl_CreateSlave(Tcl_Interp *interp,`

  `CONST charslaveName, int isSafe)`

Creates a “slave” interpreter with the name `slaveName`. The `isSafe` parameter determines whether to create a safe interpreter or not.

- `int Tcl_IsSafe(Tcl_Interp *interp)`

Returns `1` if the interpreter is safe, otherwise `0`.

- `int Tcl_MakeSafe(Tcl_Interp *interp)`

Transforms the interpreter into a safe interpreter, removing any “dangerous” commands and variables. Does not remove commands from extensions, so simply calling this function is not a guarantee of safety.

- `Tcl_Interp *Tcl_GetSlave(Tcl_Interp *interp,`

  `CONST char *slaveName)`

Returns the slave interpreter of `interp` named by `slaveName`.

- `Tcl_Interp *Tcl_GetMaster(Tcl_Interp *interp)`

Returns the master of the given slave interpreter.

- `int Tcl_GetInterpPath(Tcl_Interp *askingInterp,`

  `Tcl_Interp *slaveInterp)`

Sets the result in `askingInterp` to the path between `askingInterp` and `slaveInterp`. Returns `TCL_OK`, or `TCL_ERROR` if the path cannot be computed.

- `int Tcl_HideCommand(Tcl_Interp *interp,`

  `CONST char *cmdName, CONST char *hiddenCmdName)`

Moves `cmdName` to the set of hidden commands, giving it the name `hiddenCmdName`. If `cmdName` doesn’t exist as a visible command, `TCL_ERROR` is returned.

- `int Tcl_ExposeCommand(Tcl_Interp *interp,`
```
          CONST char *hiddenCmdName, CONST char *cmdName)
```
Moves the hidden command referred to by `hiddenCmdName` to the visible command `cmdName`. Returns `TCL_ERROR` if the command doesn't exist.

- `int Tcl_CreateAlias(Tcl_Interp *slaveInterp,`
```
          CONST char *slaveCmd, Tcl_Interp *targetInterp,
          CONST char *targetCmd, int argc,
          CONST char **argv)

  int Tcl_CreateAliasObj(Tcl_Interp *slaveInterp,
          CONST char *slaveCmd, Tcl_Interp *targetInterp,
          CONST char *targetCmd, int objc,
          Tcl_Obj **objv)
```
These two commands are essentially the same, the difference being that `Tcl_CreateAliasObj` takes an array of Tcl objects instead of strings, for the sake of efficiency. Both commands create a command `slaveCmd` in `slaveInterp`, aliasing it to `targetCmd` in `targetInterp`. The arguments in `argv` or `objv` are prefixed to any arguments passed to the command alias.

- `int Tcl_GetAlias(Tcl_Interp *interp,`
```
          CONST char *slaveCmd, Tcl_Interp *targetInterpPtr,
          CONST char *targetCmdPtr, int *argcPtr,
          CONST char ***argvPtr)

  int Tcl_GetAliasObj(Tcl_Interp *interp,
          CONST char *slaveCmd, Tcl_Interp *targetInterpPtr,
          CONST char *targetCmdPtr, int *objcPtr,
          Tcl_Obj ***objvPtr)
```
Both of these functions obtain information about an aliased command by looking up an alias `slaveCmd` in `interp` and filling in the `targetInterpPtr`, `targetCmdPtr`, a pointer to the number of arguments, and then either `objvPtr` or `argvPtr`.

31.2 Interpreters

The central data structure used by the Tcl library is a C structure of type `Tcl_Interp`. Throughout this part of the book we'll refer to these data structures as interpreters. An interpreter embodies the execution state of a Tcl script, including Tcl procedures, commands implemented in C, variables, and an execution stack that reflects the internal state of command

and script evaluation. It is, in a sense, the world in which a Tcl script is run, and Tcl scripts cannot see outside the interpreter they are in. The interpreter keeps track of where it is in the script being evaluated, what commands are available to be called, as well as the variables that have been set within it. Most of the Tcl library procedures take a pointer to a `Tcl_Interp` structure as an argument.

Tcl applications often use only a single interpreter; however, it is possible for a single process to manage several independent interpreters. As an example, consider a server that responds to requests from different clients by executing some Tcl code when it receives a network connection. Each interpreter could be responsible for executing code from a specific client, and since different interpreters are used, their data is not visible to one another, even though they reside in the same address space. Keep in mind that multiple interpreters are not multiple threads, though, and so do not resolve problems of concurrency. In order to learn about concurrency issues, see Chapter 43 on events and Chapter 46 on threads.

If you embed Tcl in your own code, you are responsible for creating one or more interpreters. If you write a Tcl extension, when it is loaded, Tcl add the extension's commands and variables to those in the interpreter in which it is loaded.

31.3 A Simple Tcl Application

Up until now you've run Tcl programs by using the `tclsh` or `wish` binaries, but of course that's not the only way.

The following program illustrates how to create and use an interpreter. It is a simple but complete Tcl application that evaluates a Tcl script in a file and prints either the results or, in the case of an error, the error message.

```
#include <tcl.h>
int main(int argc, char *argv[]) {
    Tcl_Interp *interp;
    char *result;
    int code;

    if (argc != 2) {
        fprintf(stderr,
                "Wrong number of arguments: ");
        fprintf(stderr,
                "should be \"%s filename\"\n", argv[0]);
        exit(1);
    }

    interp = Tcl_CreateInterp();
    code = Tcl_EvalFile(interp, argv[1]);
    result = Tcl_GetStringResult(interp);
    if (code != TCL_OK) {
        printf("Error was: %s\n", result);
        exit(1);
    }
    printf("Result was: %s\n", result);
    Tcl_DeleteInterp(interp);
    exit(0);
}
```

If you have a complete installation of Tcl, compiling this code should be relatively easy, something along the lines of

```
cc -o simple simple.c -ltcl8.5
```

on a Debian Linux system. Depending on your installation of Tcl, you may need to add flags such as `-I/usr/include/tcl8.5` which is necessary, for example, on Debian Linux systems. For more information on building Tcl applications and extensions, see Chapter 47.
You can now use your program to run Tcl scripts, like so:

```
./simple hello.tcl
```

Let's go back and look at `simple.c`. For this simple example, we do not need to include any headers besides `tcl.h`, which already includes many other headers in turn. Logically, whenever you use Tcl, you need to include `tcl.h`.
In the `main` function, we create a pointer to an interpreter, a return code integer, and a pointer to a result string.
We then make sure that we have an argument to our program—the file containing the Tcl script to evaluate. Next, we create the interpreter. This

new interpreter contains all the built-in commands defined in the Tcl library (`libtcl8.5.so` in this case), but few of the Tcl procedures or variables that standard `tclsh` has, because we do not load the standard initialization scripts. For information on which variables are present where, see the `tclvars` and `tclsh` reference documentation. After the file is evaluated, two pieces of information are available: first, the return code from `Tcl_EvalFile`, which is an `int` that corresponds to `TCL_OK` if everything went well, or `TCL_ERROR` if there was a problem, or any one of `TCL_RETURN`, `TCL_BREAK`, or `TCL_CONTINUE` for other various conditions. To get the result in string form, or an error string, we call `Tcl_GetStringResult` and then, depending on whether it's an actual result or an error condition, print something to that effect.
See Chapter 33 for more ways of evaluating Tcl code from your own applications and extensions.

31.4 Deleting Interpreters

When a Tcl interpreter is no longer needed, it may be deleted with the `Tcl_DeleteInterp` call. This frees the interpreter and everything associated with it, including variables, commands, and file descriptors that are not shared with other interpreters.

31.5 Multiple Interpreters

One of Tcl's most interesting features is the ability to have multiple interpreters present and active. See Chapter 15 for a discussion of manipulating multiple interpreters from Tcl scripts. As stated above, multiple interpreters are not a mechanism for concurrency; however, they are very useful for isolating code that must be run separately. The basic mechanism for creating subinterpreters (also known as *slaves*) is the `Tcl_CreateSlave` function, which is used as in the following example:

```
char *interp_names[] = {"a", "b", "c", "d", "e", "f"};
/* Port numbers to use for interpreters. */
int interp_ports[] = {10000, 10001, 10002,
                      10003, 10004, 10005};
...

for (i = 0; i < NUM_INTERPS; i++) {
    /* Create a slave interpreter. */
    slave_interps[i] = Tcl_CreateSlave(
        interp, interp_names[i], 0);
    /* Set the 'port' variable in the slave
       interpreter. */
    Tcl_SetVar2Ex(slave_interps[i], "port", NULL,
        Tcl_NewIntObj(interp_ports[i]), 0);
    /* Run the script in the new slave interpreter. */
    Tcl_EvalFile(slave_interps[i], argv[1]);
}
```

The loop creates a series of slave interpreters, each of which has a port defined on which to listen, so that the Tcl code can listen on a separate port for each one. If you are very sure of who you are connecting with, you could even evaluate code sent over the socket and return the results, and problems in one interpreter would not spill over into the others. `Tcl_CreateSlave` takes a Tcl interpreter, name, and finally an integer specifying whether to make the slave a safe interpreter or not.

32. Tcl Objects

In Tcl, all values can be treated as strings. Early versions of Tcl actually stored all values as strings, even numeric data types. This approach could be quite slow, especially for values that needed frequent conversion to binary formats, such as floating-point numbers.
In contrast, all modern versions of Tcl (ever since 8.0, released in 1997) store their data internally using a much more efficient form called *Tcl objects*. Tcl objects are values that have a string representation and a second "internal" representation of some other type, such as integer, double, list, or, in Tcl 8.5, dictionaries. Tcl uses objects internally for variable values, command arguments, command results, and scripts.
This dual nature of Tcl objects means that they are fast where they need to be, efficient in terms of memory, and still easy to work with. For example, in a `while` loop with a counter variable, the variable is transformed into an integer and remains an integer for the duration of the loop. This makes the Tcl code much more efficient than having a string that is constantly being transformed to an integer and back, each time it is incremented.
Since objects are a newer addition to Tcl, for many operations there are two API calls: one that uses objects, and one that uses the string-based interface. The older API is simpler, but at the cost of speed, especially where numbers are concerned.
Tcl objects are represented at the C level by the `Tcl_Obj` structure:

```
Tcl_Obj *obj = Tcl_NewObj();
```

This code creates a new, empty object. In most cases, you should view this as an opaque type—you don't need to deal with the structure's members, because the Tcl API does that for you.

32.1 Functions Presented in This Chapter

- `Tcl_Obj *Tcl_NewObj()`

Creates a new object.

- `Tcl_Obj *Tcl_DuplicateObj(Tcl_Obj *`*`objPtr`*`)`

Returns a new copy of the object, with reference count 0.

- `Tcl_IncrRefCount(Tcl_Obj *objPtr)`

Increases the object's reference count.

- `Tcl_DecrRefCount(Tcl_Obj *objPtr)`

Decreases the object's reference count and frees it if the reference count goes to 0 or less.

- `int Tcl_IsShared(Tcl_Obj *objPtr)`

Returns `1` if the object is shared, `0` otherwise.

- `Tcl_InvalidateStringRep(Tcl_Obj *objPtr)`

Marks the object's string representation as invalid and frees space associated with it.

- `Tcl_Obj *Tcl_NewBooleanObj(int boolValue)`

Creates a new object with an initial value `boolValue`.

- `Tcl_Obj *Tcl_NewIntObj(int intValue)`

Creates a new object with an initial value `intValue`.

- `Tcl_Obj *Tcl_NewLongObj(long longValue)`

Creates a new object with an initial value `longValue`.

- `Tcl_Obj *Tcl_NewWideIntObj(Tcl_WideInt wideValue)`

Creates a new object with an initial value `wideValue`.

- `Tcl_Obj *Tcl_NewBignumObj(mp_int *bigValue)`

Creates a new object with an initial arbitrary-precision integer value `bigValue`.

- `Tcl_Obj *Tcl_NewDoubleObj(double doubleValue)`

Creates a new object with an initial value `doubleValue`.

- `Tcl_Obj *Tcl_NewStringObj(const char *bytes, int length)`

Creates a new object with an initial value of the UTF-8 string from `bytes`. Only the first `length` bytes are copied, unless `length` is negative, in which case all bytes up to the first null character are copied.

-
  ```
  Tcl_Obj *Tcl_NewByteArrayObj(CONST unsigned char *bytes,
                int length)
  ```

Creates a new byte array object from `bytes`.

- `Tcl_SetBooleanObj(Tcl_Obj *objPtr, int boolValue)`

Sets the existing object `objPtr` to Boolean `boolValue`.

- `Tcl_SetIntObj(Tcl_Obj *objPtr, int intValue)`

Sets the existing object `objPtr` to `intValue`.

- `Tcl_SetLongObj(Tcl_Obj *objPtr, long longValue)`

Sets the existing object `objPtr` to `longValue`.

- `Tcl_SetWideIntObj(Tcl_Obj *objPtr, Tcl_WideInt wideValue)`

Sets the existing object `objPtr` to `wideValue`.

- `Tcl_SetBignumObj(Tcl_Obj *objPtr, mp_int *bigValue)`

Sets the existing object `objPtr` to the arbitrary-precision integer value *bigValue*.

- `Tcl_SetDoubleObj(Tcl_Obj *`*`objPtr`*`, double `*`doubleValue`*`)`

Sets the existing object *objPtr* to *doubleValue*.

- `Tcl_SetStringObj(Tcl_Obj *`*`objPtr`*`, const char *`*`bytes`*`,`

  `int `*`length`*`)`

Sets the existing object *objPtr* to the value of the UTF-8 string from *bytes*. Only the first *length* bytes are copied, unless *length* is negative, in which case all bytes up to the first null character are copied.

- `void Tcl_SetByteArrayObj(Tcl_Obj *`*`objPtr`*`,`

  `CONST unsigned char *`*`bytes`*`, int `*`length`*`)`

Sets the object to be a byte array containing *bytes*.

- `unsigned char *Tcl_SetByteArrayLength(Tcl_Obj *`*`objPtr`*`,`

  `int `*`length`*`)`

Sets the length of the byte array, truncating it or extending it (with arbitrary values), and returns the object's new array of bytes.

- `int Tcl_GetBooleanFromObj(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`objPtr`*`, int *`*`boolPtr`*`)`

Gets the Boolean value of *objPtr* and places it in *boolPtr*. Returns `TCL_ERROR` on failure.

- `int Tcl_GetIntFromObj(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`objPtr`*`, int *`*`intPtr`*`)`

Gets the integer value of *objPtr* and places it in *intPtr*. Returns `TCL_ERROR` on failure.

- `int Tcl_GetLongFromObj(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`objPtr`*`, long *`*`longPtr`*`)`

Gets the long value of *objPtr* and places it in *intPtr*. Returns `TCL_ERROR` on failure.

- `int Tcl_GetWideIntFromObj(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`objPtr`*`, Tcl_WideInt *`*`widePtr`*`)`

Gets the wide integer value of *objPtr* and places it in *intPtr*. Returns `TCL_ERROR` on failure.

- `int Tcl_GetBignumFromObj(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`objPtr`*`, mp_int *`*`bigValue`*`)`

Gets the arbitrary-precision integer value of *objPtr* and places it in *bigValue*. Returns `TCL_ERROR` on failure.

- `int Tcl_GetDoubleFromObj(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`objPtr`*`, double *`*`doublePtr`*`)`

Gets the double value of *objPtr* and places it in *doublePtr*. Returns `TCL_ERROR` on failure.

- `char *Tcl_GetStringFromObj(Tcl_Obj *`*`objPtr`*`,`

```
                int *lengthPtr)
```

Returns a pointer to the object's string representation and places the length of the string in `lengthPtr` if it is non-null.

- `unsigned char *Tcl_GetByteArrayFromObj(Tcl_Obj *objPtr,`

```
                int *lengthPtr)
```

Returns the bytes contained in a byte array object and places the length of the bytes in `lengthPtr` if it is non-null.

- `char *Tcl_Alloc(int size)`

```
  char *Tcl_AttemptAlloc(int size)
  char *ckalloc(int size)
  char *attemptckalloc(int size)
```

Returns a pointer to a block of at least `size` bytes suitably aligned for any use. Use these routines rather than the native `malloc`. `Tcl_AttemptAlloc` does not cause the Tcl interpreter to panic if the memory allocation fails, whereas `Tcl_Alloc` does. `ckalloc` and `attemptckalloc` are macros equivalent to their corresponding functions, except they also support advanced memory debugging. See the reference documentation for more information.

- `char *Tcl_Realloc(char *ptr, int size)`

```
  char *Tcl_AttemptRealloc(char *ptr, int size)
  char *ckrealloc(char *ptr, int size)
  char *attemptckrealloc(char *ptr, int size)
```

Changes the size of the block pointed to by `ptr` to `size` bytes and returns a pointer to the new block. The contents are unchanged up to the lesser of the new and old sizes. The returned location may be different from `ptr`. `Tcl_AttemptRealloc` does not cause the Tcl interpreter to panic if the memory allocation fails, whereas `Tcl_Realloc` does. `ckrealloc` and `attemptckrealloc` are macros equivalent to their corresponding functions, except they also support advanced memory debugging.

- `Tcl_Free(char *ptr)`

```
  ckfree(char *ptr)
```

Makes the space referred to by `ptr` available for further allocation. `ckfree` is an equivalent macro that also supports advanced memory debugging.

32.2 String Objects

Since all Tcl objects may be represented as strings, we will begin our exploration of Tcl objects with them.
To create a new string object:

```
Tcl_Obj *strobj;
strobj = Tcl_NewStringObj("Tcl!", -1);
```

A new string object is created from a series of bytes (in the UTF-8 encoding—see Chapter 39 for more information about encodings) and their length. In this case, the `-1` value means that we let Tcl determine the string length instead of specifying the value ourselves, which is often a handy shortcut.
Strings and commands to manipulate them are covered in more detail in Chapter 39.

32.3 Numerical Objects

Creating an object that contains an integer is similar:

```
Tcl_Obj *intobj;
intobj = Tcl_NewIntObj(42);
```

This version of `intobj` is far more efficient to use in mathematical operations than a string representation of the number 42, because it contains an actual integer as its internal representation.
Other numerical types that can be created are the following:

- Boolean—Either true or false, `1` or `0`. Created with `Tcl_NewBooleanObj(`*`int`*`)`.
- long—Holds a long integer. Created with `Tcl_NewLongObj(`*`long`*`)`.
- wide—Holds a “wide” (64-bit) integer. Created with `Tcl_NewWideIntObj(`*`wideValue`*`)`, where *`wideValue`* is at least a 64-bit value.
- bignum—Holds an arbitrary-precision integer. Created with `Tcl_NewBignumObj(`*`bigValue`*`)`, where *`bigValue`* is a pointer to an `mp_int` structure as declared by the LibTomMath arbitrary-precision integer library.
- double—Contains a C double type. This is the type Tcl uses to deal with floating-point numbers. Created with `Tcl_NewDoubleObj(`*`double`*`)`.

32.4 Fetching C Values from Objects

To use a Tcl object in your C code, you’ll need to retrieve its underlying value, whether a string, an integer, a double, or some other format. Your program asks the Tcl library to translate a given object into the type you need. This may be directly accessible from the string or internal representation, or some conversion may need to be done (e.g., retrieving a double value from an object having an integer internal representation). While Tcl guarantees that any arbitrary object can be retrieved as a string, this is not true for other data types. Note also that by asking for a value as a specific type, Tcl modifies the internal representation to be that type—if, of course, it was possible to perform the requested fetch. For example, if an object contains an integer internal representation, and `Tcl_GetDoubleFromObj` is called on it successfully, the new internal representation is a double.

It is always possible to transform an object into a string, using the `Tcl_GetStringFromObj` call, but it may not necessarily be possible to transform an object into other types, such as integer, double, and so on. For this reason, `Tcl_GetIntFromObj` and the other calls to obtain various types from Tcl objects take an interpreter as an argument, so that they may raise an exception should they encounter a problem transforming the object into the desired type.

```
char *intstring;
intstring = Tcl_GetStringFromObj(intobj);
/* No problem. */

int strint;

Tcl_Obj *strobj;
strobj = Tcl_NewStringObj("Tcl!", -1);
Tcl_GetIntFromObj(interp, strobj, &strint);
/* Fails! */
```

In the first case, the call to `Tcl_GetStringFromObj` is always successful, because by their very nature, Tcl objects can always be represented as strings —`intstring` is `42`. However, the call to `Tcl_GetIntFromObj` does return an error, because we attempt to call it with the object `strobj`. This object contains the string `Tcl!`, which cannot be transformed into an integer, and so a `TCL_ERROR` result is returned, and an error message is left in the interpreter. As with object creation, functions exist for all types in order to attempt to extract the type in question from the Tcl object—for example, `Tcl_GetLongFromObj`, `Tcl_GetDoubleFromObj`, and so on for the numerical types.

32.5 The Dynamic Nature of a Tcl Object

To illustrate the dual nature of a Tcl object, let’s examine its life in a Tcl script:

```
1. set foo 35100
2. puts "The value of \$foo is: $foo"
3. incr foo 100
4. puts "Now \$foo is: $foo"
```

On line 1, the variable `foo` is set to an object containing the characters `35100`. At this point, the object is a string.
When the script on line 2 prints the variable `foo`, its string value is needed, and `35100` is used.
The object that `foo` contains is transformed in line 3. Whereas it was previously a string, we now wish to increment its value by 100, so we need to deal with it as a number. As it is not currently a number, it is set to the integer type, and its internal representation is set to 35100. The `incr` command then acts on this integer, incrementing it to 35200. The string representation of the object is marked as invalid, but a new string representation is not generated at this time; regenerating the string representation could be wasted effort if the object is changed by further arithmetic operations.
On the final line, we print the value of `foo` again. Its string representation is not currently valid, so we update it to `35200`, which the `puts` command then prints. At the end of the script, the object contains both a valid string representation and the integer 35200.

32.6 Byte Arrays

Byte arrays exist to hold arbitrary sequences of 8-bit values that do not necessarily correspond to characters as string objects must. Byte arrays are useful for holding binary data—such as a JPEG or an MP3 sound file—that might contain embedded nulls.

```
unsigned char jpegheader[11] =
    {0xff, 0xff, 0xd8, 0xff, 0xe0, 0x00,
     0x10, 0x4a, 0x46, 0x49, 0x46};
Tcl_Obj *byteobj;
byteobj = Tcl_NewByteArrayObj(jpegheader, 11);
```

This example creates a Tcl object that represents the first bytes of a JPEG file header.

32.7 Composite Objects

Prior to Tcl 8.4, Tcl had one type of object that could contain other objects: a list. From the perspective of C, lists are Tcl objects that contain an array of other Tcl objects. Manipulating lists from a Tcl script is discussed in Chapter 6. Tcl's C API lets you do everything you can do from Tcl and is covered in further detail in Chapter 41.
As of version 8.5 (although there is also a backport available for 8.4), Tcl also provides dictionary objects (also referred to as dict objects) that reference other objects through a hash table. The Tcl script commands for interacting with dictionary objects are discussed in Chapter 7. The C interface allows you to do many of the same things and is covered together with lists in Chapter 41.
Tcl lists and dictionaries are good ways of passing more complex structures back and forth between Tcl and C, as we saw in the section on representing information in Chapter 30. Lists are best suited for situations where elements are accessed via a numerical index (fetch element 5, for instance), whereas dictionaries are efficient when looking up an object based on a string describing it, for instance, mapping "Rome" to "Italy," "Paris" to "France," "Berlin" to "Germany," and so on.

32.8 Reference Counting

To avoid creating unnecessary copies of an object, many parts of your application may share the same Tcl object at the same time. For example, a single Tcl object might be used in a procedure, stored in a variable, used within the Tcl library itself, and so on. In order to manage objects and the memory they consume, Tcl uses a technique called *reference counting*. Reference counting solves the same basic problem as garbage collection in languages like Scheme, Ruby, or Java. It is robust, fast, straightforward, and portable, and it makes working with Tcl at the C level simple.
A newly created Tcl object has an initial reference count of 0. Any code that needs to mark the object as being in use should increment the reference count of the object using the `Tcl_IncrRefCount` function. When the code no longer needs the object, it should decrement the reference count with `Tcl_DecrRefCount`. Once the reference count of an object is decremented to 0 or less, Tcl can free the object.

Many Tcl functions temporarily modify the reference counts of object that you provide as arguments. If your code creates a new object (with a reference count of 0), and then passes it to a Tcl function without first incrementing its reference count, the Tcl function could internally increment and later decrement the reference counter of the object you provided. The function's internal decrement of the object's reference count back to 0 would trigger the deletion of the object. Subsequent attempts to use the object in your code would of course result in an error.

Generally speaking, you should increment the reference count of any newly created object and then decrement the reference count when the pointer variable goes out of scope. However, there are a few Tcl functions that store a new reference to an existing object that you provide, incrementing the object's reference count. These functions handle putting an object in a variable, storing it in a list or dictionary object, or putting it in the interpreter's result. (All of these actions are discussed in subsequent chapters.) If you create an object only to pass it to one of these functions, it's safe to omit the `Tcl_IncrRefCount`/`Tcl_DecrRefCount` of the object.

32.9 Shared Objects

Tcl implements a "copy-on-write" system for objects. This means that an object may be shared among several variables, for instance, as long as none of them is written to. Once a write occurs, a copy of the object is made. What this signifies when you are writing C code for Tcl is that when directly modifying an object, you need to check to ensure that it's not shared via the `Tcl_IsShared` function, and if it is shared, copy it using `Tcl_DuplicateObj`:

```
if (Tcl_IsShared(objPtr)) {
    objPtr = TclDuplicateObj(objPtr);

    /* objPtr now points to a duplicate of the original
     * object, with a reference count of 0.
     */
}
```

Commands that need this check are usually those that access and modify an object that's the value of a variable, like `incr` and `lappend`. This bit of Tcl code illustrates an object that is at first shared and subsequently copied when modified:

```
set foo 1
set bar $foo;
# foo and bar share the same object - '1'.
incr bar;
# bar now has its own object - '2'.
```

Initially, an object containing `1` is created and assigned to the `foo` variable. Then, the same object is assigned to the `bar` variable. At this point, both variables refer to one object, whose value is `1`. The next command, `incr bar`, notices that the value is shared, makes a copy, and increments the value of the copy. The `foo` variable now contains an object with the value `1`, and `bar` contains an integer object with the value `2`. If we were to evaluate `incr bar` a second time, `Tcl_IsShared` would return false, because the object is now distinct from that contained in `foo`.

32.10 New Object Types

Since they are rarely needed, we won't cover them in much depth here, but you should be aware that if the need arises, it is possible to create new object types via Tcl's C API.

Creating a new object type makes sense where the object is easily represented as a string that may be converted to the type in question and then back again without losing information. For more complex C data structures, a better approach is the technique of mapping the structure to a unique tag via a hash table, mentioned in Section 30.4.

We must also ensure that if a user modifies the object value as a string, nothing bad will happen if we transform it back to the type in question. Consider what would happen if we were to create a Tcl "reference" object type that contained a C pointer, say, `0x10001800`. If the user modified it, as a string, to be `)0x10001810`, and then tried to use it again as a pointer type, the most likely result would be a segmentation fault or data corruption.

The most sensible reason to create a new Tcl object type is for simple (not represented as a complex structure in C) new data types that need to efficiently represent information and don't have problems representing it as a string.

A good example of how to create a new data type is provided by Salvatore Sanfilippo's `tclsbignum` package, which provided arbitrary-precision integer math capabilities prior to Tcl 8.5's native `BignumObj` support.

The core of a new type is the `Tcl_ObjType` structure:

```
typedef struct Tcl_ObjType {
    char *name;
    Tcl_FreeInternalRepProc *freeIntRepProc;
    Tcl_DupInternalRepProc *dupIntRepProc;
    Tcl_UpdateStringProc *updateStringProc;
    Tcl_SetFromAnyProc *setFromAnyProc;
} Tcl_ObjType;
```

When creating a new type, the programmer supplies functions that

- Free the storage of the type via the `Tcl_FreeInternalRepProc` slot. In the bignum library, this just means freeing the associated memory.
- Make a copy of the object via the `Tcl_DupInternalRepProc` slot. If the type is a data structure, it is up to the programmer to decided whether to make a "deep copy" or not. By deep copy we mean copying the entire data structure, not just the top, or head. For instance, when copying a list, you could make a copy of the list structure itself and leave the elements pointing to the same objects, or you could make copies of all the elements as well (and so on recursively if the elements are in turn lists). This is a deep copy. In reality, Tcl list copies are not deep copies. In the bignum library, the bignum value is copied.
- Update the object's string representation to match the current value of the object via the `Tcl_UpdateStringProc` slot. The bignum library's string update function fills in the string representation with a base 10 representation of the number.
- The opposite transformation—re-creating the internal representation from a string—is performed by the `SetFromAnyProc` function slot in the `Tcl_ObjType` structure. In this case, the bignum library must parse the string into a number, or return an error if it is unable to do so.

32.11 Parsing Strings

Prior to the advent of Tcl objects, when everything was represented as a string internally, it was necessary to be able to easily obtain other types from those strings by parsing them. Tcl still provides those calls, and they are occasionally handy when working with strings:

```
int Tcl_GetInt(Tcl_Interp *interp, CONST char *string,
    int *intPtr);
int Tcl_GetDouble(Tcl_Interp *interp,
    CONST char *string, int *doublePtr);
int Tcl_GetBool(Tcl_Interp *interp, CONST char *string,
    int *intPtr);
```

These calls all take an interpreter and a string containing the information to be parsed as arguments, as well as a pointer to a C variable of the desired type, with a Boolean being either `1` or `0`. On success, they return `TCL_OK`, and on failure, `TCL_ERROR`; for instance:

```
int num = 0;
Tcl_GetInt(interp, "42", &num);
```

This code sets `num` to `42` and returns `TCL_OK`.

32.12 Memory Allocation

Tcl provides, as you will see in later chapters, many functions to perform common actions in a uniform, cross-platform manner. One of the most important activities for which Tcl provides a layer is memory allocation. To allocate and free memory in Tcl, we use `Tcl_Alloc` and `Tcl_Free`. These functions are equivalent to `malloc` and `free` in C, but be careful not to mix and match them, such as calling `Tcl_Free` on memory allocated with `malloc` or `strdup`, because on some platforms with certain Tcl configurations, this causes problems. Additionally, Tcl provides the `Tcl_Realloc` function, to use instead of `realloc`, to change the size of an allocated block of memory. The location returned may be different from the original location, but the contents of the block are unchanged up to the lesser of the old and the new sizes.
`Tcl_Alloc` and `Tcl_Realloc` cause the Tcl interpreter to panic if the memory allocation fails. To avoid this, you can call `Tcl_AttemptAlloc` and `Tcl_AttemptRealloc` instead, which simply return `NULL` if they fail.
Corresponding to all of these functions is a set of macros: `ckalloc`, `ckfree`, `ckrealloc`, `attemptckalloc`, and `attemptckrealloc`. Normally, they are synonyms for the corresponding procedures described previously. However, when Tcl and all modules calling Tcl are compiled with `TCL_MEM_DEBUG` defined, these macros are redefined to be special debugging versions of these procedures. This enables the `memory` ensemble for tracking and reporting memory allocation. See the `memory` reference documentation for more information. If

you anticipate requiring memory debugging of your code, you should use these macros instead of direct calls to `Tcl_Alloc` and the other functions.

33. Evaluating Tcl Code

This chapter illustrates how to evaluate Tcl scripts from your C code, as well as Tcl math expressions.

33.1 Functions Presented in This Chapter

- `int Tcl_EvalObjEx(Tcl_Interp *interp,`
  `Tcl_Obj *objPtr, int flags)`

Evaluates a Tcl script contained in an object.

- `int Tcl_EvalFile(Tcl_Interp *interp,`
  `CONST char *fileName)`

Evaluates a Tcl script in a file given by `fileName`.

- `int Tcl_EvalObjv(Tcl_Interp *interp, int objc,`
  `Tcl_Obj **objv, int flags)`

Executes a single preparsed command instead of a script. The `objc` and `objv` arguments contain the values of the words for the Tcl command, one word in each object in `objv`.

- `int Tcl_Eval(Tcl_Interp *interp, CONST char *script)`

Evaluates `script`.

- `int Tcl_EvalEx(Tcl_Interp *interp,`
  `CONST char *script, int numBytes, int flags)`

Evaluates `script`. More efficient than `Tcl_Eval`.

- `int Tcl_GlobalEval(Tcl_Interp *interp,`
  `CONST char *script)`

Evaluates `script` in the global namespace. Deprecated.

- `int Tcl_GlobalEvalObj(Tcl_Interp *interp,`
  `Tcl_Obj *objPtr)`

Evaluates the script in `objPtr` in the global namespace. Deprecated.

- `int Tcl_VarEval(Tcl_Interp *interp, char *string,`
  `char *string, ... (char *) NULL)`

Takes any number of string arguments of any length, concatenates them into a single string, then evaluates that string. Deprecated.

- `int Tcl_VarEvalVA(Tcl_Interp *interp,`
  `va_list argList)`

Like `Tcl_VarEval`, but takes a `va_list` instead of multiple strings. Deprecated.

- `Tcl_Obj *Tcl_GetObjResult(Tcl_Interp *interp)`

Returns the result for `interp` as an object. The object's reference count is not incremented.

- `const char *Tcl_GetStringResult(Tcl_Interp *interp)`

Returns the result for `interp` as a string.

33.2 Evaluating Tcl Code

In Chapter 31 we saw the use of `Tcl_EvalFile` to evaluate a Tcl script in a file. Tcl provides several other functions for evaluating scripts. These functions all take an interpreter as their first argument, return a completion code, and set a result in the interpreter. The most direct of these is `Tcl_Eval`:

```
Tcl_Obj *result;
char script[] = "set a [expr {20 * 30}]";
...
code = Tcl_Eval(interp, script);
result = Tcl_GetObjResult(interp);
```

This simply evaluates the string passed to it. If it's successful, it returns `TCL_OK`, and on failure, it returns `TCL_ERROR` as well as setting a result in the interpreter.
You can use the `Tcl_GetObjResult` function to retrieve a pointer to the Tcl object containing the result. The object's reference count is not incremented; use `Tcl_IncrRefCount` to increment its reference count if you need to retain a long-term pointer to the object. For legacy code, `Tcl_GetStringResult` returns the result as a string; you should use `Tcl_GetObjResult` for new code development.

Note

Because of the way Tcl works internally in versions prior to 8.4, it may need to make some temporary modifications to the string while it is working with it, so it is important that you pass a modifiable string to `Tcl_Eval` rather than pass it a string constant directly. Code such as `Tcl_Eval(interp, "your script")` may fail!

If you want to guarantee maximum efficiency from your script, consider using `Tcl_EvalObjEx`. In addition to using the object interface, Tcl also byte-compiles and caches the code passed to it, making it significantly faster should it be executed again at some point in the future.

```
Tcl_Obj *script = Tcl_NewStringObj(
    "while { $i < 10 } { .... }", -1);
Tcl_IncrRefCount(script);
Tcl_EvalObjEx(interp, script, 0);
...
```

This example creates a new string object that contains the script we wish to evaluate, increases its reference count, and then evaluates it. The `0` in the call to `Tcl_EvalObjEx` is its "flags" argument. The two valid flags are `TCL_EVAL_DIRECT`, which instructs Tcl not to byte-compile the script, and `TCL_EVAL_GLOBAL`, which ensures that the script is processed at the global level, running in the global namespace and using only global variables.
Each call to a procedure such as `Tcl_Eval` must contain a complete script. The following does not work:

```
char part1[] = "set ";
char part2[] = "a ";
char part3[] = "32";
Tcl_Eval(interp, part1); /* Fails. */
Tcl_Eval(interp, part2);
Tcl_Eval(interp, part3);
```

This code fails at the first call to `Tcl_Eval`, because the script it is attempting to evaluate is not complete.

33.3 Dynamically Building Scripts

There are several ways to compose a script from multiple pieces. The simplest is to build the command up as a list, in order to properly quote whitespace within individual elements:

```
Tcl_Obj *cmd;
cmd = Tcl_NewObj();
Tcl_ListObjAppendElement(interp, cmd,
    Tcl_NewStringObj("puts", -1));
Tcl_ListObjAppendElement(
    interp, cmd, Tcl_NewStringObj("hello world", -1));
Tcl_IncrRefCount(cmd);
Tcl_EvalObjEx(interp, cmd, 0);
Tcl_DecrRefCount(cmd);
```

Note that because we build the command up as a list, with

Tcl_ListObjAppendElement, we don't have to quote the hello world string ourselves, even though it contains a space. When Tcl evaluates the object, it is evaluating an object akin to puts {hello world}. Using lists is also more efficient than using strings. If one of the elements is a number, for example, it can be added to the list as a number instead of the number's string representation.

Another way of dynamically creating a command to evaluate is by using the Tcl_EvalObjv function. It takes an array of Tcl objects and a count of the objects as arguments, in addition to an interpreter:

```
int i = 0;
Tcl_Obj *scriptArr[3];
scriptArr[0] = Tcl_NewStringObj("set", -1);
scriptArr[1] = Tcl_NewStringObj("a", -1);
scriptArr[2] = Tcl_NewIntObj(32, -1);
for (i = 0; i < 3; i++) {
   /* Increase the reference counts. */
   Tcl_IncrRefCount(scriptArr[i]);
}
Tcl_EvalObjv(interp, scriptArr, 3, 0);
for (i = 0; i < 3; i++) {
    /* Decrease the reference counts. */
    Tcl_DecrRefCount(scriptArr[i]);
}
```

In this code, we create the array of objects, fill it in, increment the reference counts, perform the evaluation, and then lower the reference counts again.

33.4 Tcl Expressions

In addition to evaluating Tcl code, you can also calculate the result of a Tcl expression. Expressions are useful as conditions for commands like if and while, or as a way to perform some math with Tcl variables. See Chapter 4 on the expr command for further information on the syntax and utility of expressions.

The basic method of performing an expression evaluation is via functions like Tcl_ExprTYPE and Tcl_ExprTYPEObj, where TYPE is one of Long, Double, or Boolean, and the result is stored in a variable of the corresponding C type; for instance:

```
long result = 0;
Tcl_Obj *expr = Tcl_NewStringObj("($foo / 2) + 1", -1);
Tcl_IncrRefCount(expr);
Tcl_ExprLongObj(interp, expr, &result);
Tcl_DecrRefCount(expr);
```

This code creates a string object from the string `($foo / 2) + 1`, evaluates it, and stores the result in the result variable.
If you don't need the result returned as a particular C type, there are two generic functions for evaluating expressions: `Tcl_ExprObj` and `Tcl_ExprString`. The string form is the simplest, but it sacrifices some speed because the result is stored as a string.

```
char *result = NULL;
char *exprstr = "2 + 2";
Tcl_ExprString(interp, exprstr);
result = Tcl_GetStringResult(interp);
```

This calculates `2 + 2` and evaluates to a string result of `4`. You can then fetch the result with a call to `Tcl_GetStringResult`.
In the following we use the object-based call `Tcl_ExprObj`:

```
Tcl_Obj *result;
Tcl_Obj *expr = Tcl_NewStringObj("($foo / 2) + 1", -1);
Tcl_IncrRefCount(expr);
Tcl_ExprObj(interp, expr, &result);
Tcl_DecrRefCount(expr);
...
...
Tcl_DecrRefCount(result);
/* You are responsible for lowering the reference count of the
result. */
```

The object API is a bit more complex to use, but it is faster because it stores the result as a Tcl object with some type of numeric representation, rather than converting it back to a string.

34. Accessing Tcl Variables

In this chapter we cover how to set Tcl variables from C, how to read their values, and how to delete them. We also describe how to link C variables and Tcl variables, so that when one changes, the other changes as well. Also included is a description of variable traces.

34.1 Functions Presented in This Chapter

This chapter covers the following functions for manipulating Tcl variables from C code:

- `Tcl_Obj *Tcl_SetVar2Ex(Tcl_Interp *interp,`
  `CONST char *name1, CONST char *name2,`
  `Tcl_Obj *newValuePtr, int flags)`

Sets the variable given by `name1` (and `name2` in the case of an array element) to `newValuePtr`.

- `CONST char *Tcl_SetVar(Tcl_Interp *interp,`
  `CONST char *varName, CONST char *newValue,`
  `int flags)`

Sets the variable `varName` to `newValue`.

- `CONST char *Tcl_SetVar2(Tcl_Interp *interp,`
  `CONST char *name1, CONST char *name2,`
  `CONST char *newValue, int flags)`

Sets the variable given by `name1` (and `name2` in the case of an array element) to `newValue`.

- `Tcl_Obj *Tcl_ObjSetVar2(Tcl_Interp *interp,`
  `Tcl_Obj *part1Ptr, Tcl_Obj *part2Ptr,`
  `Tcl_Obj *newValuePtr, int flags)`

Sets the variable given by Tcl objects `part1Ptr` (and `part2Ptr` in the case of an array element) to `newValuePtr`.

- `Tcl_Obj *Tcl_GetVar2Ex(Tcl_Interp *interp,`
  `CONST char *name1, CONST char *name2, int flags)`

Retrieves the contents of the variables given by `name1` and `name2` as a Tcl object.

- `CONST char *Tcl_GetVar(Tcl_Interp *interp,`
  `CONST char *varName, int flags)`

Retrieves the contents of the variable `varName` as a string.

- `CONST char *Tcl_GetVar2(Tcl_Interp *interp,`
  `CONST char *name1, CONST char *name2, int flags)`

Retrieves the contents of the variables given by `name1` and `name2` as a string.

- `Tcl_Obj *Tcl_ObjGetVar2(Tcl_Interp *interp,`
  `Tcl_Obj *part1Ptr, Tcl_Obj *part2Ptr, int flags)`

Retrieves the contents of the variables given by Tcl objects `part1Ptr` and `part2Ptr` as a Tcl object.

- `int Tcl_UnsetVar(Tcl_Interp *interp, CONST char *varName,`
  `int flags)`

Unsets the variable `varName`.

- `int Tcl_UnsetVar2(Tcl_Interp *interp, CONST char *name1,`
  `CONST char *name2, int flags)`

Unsets the variables given by `name1` and `name2`.

- `int Tcl_TraceVar(Tcl_Interp *interp,`
  `CONST char *varName, int flags,`
  `Tcl_VarTraceProc proc, ClientData clientData)`

Arranges to call the function `proc` when the trace on `varName` matches one of the conditions specified in `flags`.

- `Tcl_UntraceVar(Tcl_Interp *interp,`
  `CONST char *varName, int flags,`
  `Tcl_VarTraceProc proc, ClientData clientData)`

Unsets a trace. Arguments must match those used to set the trace.

- `ClientData clientData Tcl_VarTraceInfo(Tcl_Interp *interp,`
  `CONST char *varName, int flags,`
  `Tcl_VarTraceProc proc, ClientData prevClientData)`

Returns the `clientData` for a previously set trace.

- `int Tcl_TraceVar2(Tcl_Interp *interp,`
  `CONST char *name1, CONST char *name2, int flags,`
  `Tcl_VarTraceProc proc, ClientData clientData)`
  `Tcl_UntraceVar2(Tcl_Interp *interp,`
  `CONST char *name1, CONST char *name2, int flags,`
  `Tcl_VarTraceProc proc, ClientData clientData)`
  `ClientData Tcl_VarTraceInfo2(Tcl_Interp *interp,`
  `CONST char *name1, CONST char *name2, int flags,`
  `Tcl_VarTraceProc proc, ClientData prevClientData)`

These functions are the same as the previous three, but instead of taking one variable name as an argument, they take a `name1` and `name2`, where `name2` is used to refer to array elements.

34.2 Setting Variable Values

Tcl variables differ from Tcl objects in that an object is just some value that can be used by Tcl, whereas a variable is an object that is accessible at the script level via a variable name.
The simplest way to create a Tcl variable is with the `Tcl_SetVar` command:

```
Tcl_SetVar(interp, "foo", "42", 0);
/* Sets the value of the foo variable to "42". */
```

Both the variable name and its value are given as strings, meaning that this is not an efficient way to create variables with numeric values. If speed is not an issue, this is the easiest way to get the job done. The variable can be either a normal variable or an array element—such as `foo` or `foo(bar)`. Other than this, it is not parsed at all, and no substitutions occur. You could use the string `[$a]` as the variable name if you so chose, although you should not use names like this because they make your code confusing and difficult to read.
The object-based version of this command is as follows:

```
Tcl_Obj *intObj;
intObj = Tcl_NewIntObj(42);
Tcl_IncrRefCount(intObj);
Tcl_SetVar2Ex(interp, "foo", NULL, intObj, 0);
Tcl_DecrRefCount(intObj);
...
```

When this code snippet is executed, it creates a `foo` variable, as before. The difference is that its value is an object containing the number 42, which is more efficient to work with if the commands that subsequently operate on `foo` deal with numbers.
The `Tcl_SetVar2Ex` function also lets us create array elements by using both the second and third arguments, like so:

```
Tcl_SetVar2Ex(interp, "foo", "bar", intObj, 0);
```

This creates the `foo(bar)` array element.
The last argument to most variable functions—the `0` in the preceding examples—is an ORed combination of flag bits:

- `TCL_GLOBAL_ONLY`

Looks up the variable in the global namespace only, even if execution is deep within nested procedures.

- `TCL_NAMESPACE_ONLY`

Looks up the variable in the current namespace only.

- `TCL_LEAVE_ERR_MSG`

Sets the interpreter's result to an error if a problem occurs when creating the variable. Otherwise, most of the set commands simply return a `NULL` value. This may prove useful within commands that should fail if they are not able to set the variable in question.

- `TCL_APPEND_VALUE`

Instead of replacing an existing value of this variable, appends to it. If the variable does not exist, it is created as normal.

- `TCL_LIST_ELEMENT`

The new value for the variable is converted to a list element (quoted, if necessary). Depending on whether the `TCL_APPEND_VALUE` flag is set, the variable is either set to a list containing this element, or the element is appended to the value contained in the variable.
In addition to the `Tcl_SetVar` and `Tcl_SetVar2Ex` functions just described, the Tcl API provides `Tcl_ObjSetVar2`, which takes Tcl objects for the variable name and/or array element name. For example, to create a list of numbers:

```
int i = 0;
Tcl_Obj *intObj;
...
for (i = 1; i <= 10; i++) {
    intObj = Tcl_NewIntObj(i);
    Tcl_IncrRefCount(intObj);
    Tcl_SetVar2Ex(interp, "numlist", NULL, intObj,
                  TCL_LIST_ELEMENT | TCL_APPEND_VALUE);
    Tcl_DecrRefCount(intObj);
}
```

In Tcl, printing the `numlist` variable would result in

```
  puts $numlist
⇒ 1 2 3 4 5 6 7 8 9 10
```

34.3 Reading Variables

The "read" that corresponds to the `Tcl_SetVar` "write" is `Tcl_GetVar`:

```
char *value;
value = Tcl_GetVar(interp, "foo", 0);
```

This returns a string pointer with the value contained in the variable `foo`. The string returned belongs to Tcl, so if you wish to keep it or modify it, make a copy with `strdup`. Calling `Tcl_SetVar` or `Tcl_SetVar2` for the variable invalidates the pointer, so copy it immediately if you want to save it.
The flags for the various `GetVar` operations are `TCL_GLOBAL_ONLY` and `TCL_LEAVE_ERR_MSG`, which have the same meaning as for the `SetVar` functions.
The object-based `GetVar` function is `Tcl_GetVar2Ex`, which returns a Tcl object:

```
Tcl_Obj *value;
value = Tcl_GetVar2Ex(interp, "foo", NULL, 0);
```

With a `Tcl_Obj` in hand, you could use `Tcl_GetIntFromObj` to get the integer value and then manipulate it, which is much more efficient than parsing the string `42` to obtain its integer value:

```
int intval = 0;
if (Tcl_GetIntFromObj(interp, value, &intval) != TCL_OK) {
    return TCL_ERROR;
}
intval = intval * 42;
```

34.4 Unsetting Variables

It is possible to remove a variable using the `Tcl_UnsetVar` or `Tcl_UnsetVar2` commands. Either

```
Tcl_UnsetVar(interp, "foo(bar)", 0);
```

or

```
Tcl_UnsetVar2(interp, "foo", "bar", 0);
```

removes the `bar` element from the `foo` array. These functions have the same effect as the Tcl `unset` command. The Tcl equivalent to this code is

```
unset foo(bar)
```

These functions return `TCL_OK` on success and `TCL_ERROR` if the variable doesn't exist or can't be removed for some other reason. The `TCL_GLOBAL_ONLY` and `TCL_LEAVE_ERR_MSG` flags may be used with these calls. If an array name is given, and no element is specified, the entire array is removed.

34.5 Linking Tcl and C Variables

Tcl provides a mechanism called *variable linking* that allows you to associate a Tcl variable with a C variable. Whenever the Tcl variable is read, the value is supplied from the C variable, and whenever the Tcl value is written, the new value is stored in the C variable as well. The function `Tcl_LinkVar` creates a link. Consider the following C code:

```
int value = 32;
...
Tcl_LinkVar(interp, "x", (void *) &value, TCL_LINK_INT);
```

This links the Tcl variable `x` to the C variable `value`. Whenever the Tcl variable `x` is read, Tcl converts `value` to a decimal string and returns the string as the value of `x` (`32` in the example). If `value` is modified, the new value is returned the next time the Tcl variable `x` is read. Whenever the Tcl variable `x` is written, `x`'s new value is converted from a Tcl object to an integer and stored in `value`. If a Tcl script attempts to write a value into `x` that isn't a proper integer, the write is rejected with an error:

```
set x "oops!"
⇒ can't set "x": variable must have integer value
```

The last argument to `Tcl_LinkVar` indicates the type of the C variable. In the previous example, the type is `TCL_LINK_INT`, which indicates that the C variable is an integer and only proper integer values may be stored in the Tcl variable. Table 34.1 lists the allowed values for the type argument and the C type to which they correspond. For any of the numerical types supported by `Tcl_LinkVar`, any attempt to assign a value to the Tcl variable that cannot be converted to the specified type (for example, a non-numerical value or an out-of-range value) is rejected with a Tcl error.

Table 34.1 Allowed Type Arguments for `Tcl_LinkVar`

Type flag	C type
TCL_LINK_INT	int
TCL_LINK_UINT	unsigned int
TCL_LINK_CHAR	char
TCL_LINK_UCHAR	unsigned char
TCL_LINK_SHORT	short
TCL_LINK_USHORT	unsigned short
TCL_LINK_LONG	long
TCL_LINK_ULONG	unsigned long
TCL_LINK_DOUBLE	double
TCL_LINK_FLOAT	float
TCL_LINK_WIDE_INT	Tcl_WideInt
TCL_LINK_WIDE_UINT	Tcl_WideUInt
TCL_LINK_BOOLEAN	int
TCL_LINK_STRING	char *

For the `TCL_LINK_STRING` type, if the C variable's value is not `NULL`, it must point to a string allocated with `Tcl_Alloc` or `ckalloc`. When the Tcl variable is modified, the old string is freed and a new one is allocated to hold the variable's new value. The Tcl variable may be assigned any string value. The Tcl variable returns the string `NULL` if it is read and the C variable is `NULL`.

Note

Only C variables allocated globally or dynamically, rather than those from the stack, should be linked.

For example, to link to a string, you would do this:

```
char *foo = "Hello, World";
/* Note the pointer to the pointer. */
Tcl_LinkVar(interp, "hi", (void *) &foo, TCL_LINK_STRING);
```

The flag `TCL_LINK_READ_ONLY` may also be ORed with the type, as in the following example:

```
Tcl_LinkVar(interp, "x", (void *), &value,
      TCL_LINK_INT|TCL_LINK_READ_ONLY);
```

This makes the Tcl variable read-only: any attempt to modify the variable from Tcl is rejected with an error.
While the Tcl variable is linked, if it is unset, Tcl automatically re-creates the variable. The Tcl variable cannot be unset permanently until the link is removed. The function `Tcl_UnlinkVar` removes a variable link previously established by `Tcl_LinkVar`. For example, the following statement removes the link created previously:

```
Tcl_UnlinkVar(interp, "x");
```

34.6 Setting and Unsetting Variable Traces

Variable traces allow you to specify a C function to be called whenever a variable is read, written, or unset. Traces can be used for many purposes. In Tk, for instance, you can configure a button widget so that it displays the value of a variable and updates itself automatically when the variable is modified. This feature is implemented with variable traces. You can also use traces for debugging, to create read-only variables, and for many other purposes.

Note

Although they are quite powerful, you should be careful in your use of traces, because they make it easy to create “magic variables” that, to the user, appear to behave very strangely.

The `Tcl_TraceVar` and `Tcl_TraceVar2` calls create variable traces, as in the following example:

```
Tcl_TraceVar(interp, "x", TCL_TRACE_WRITES,
      WriteProc, (ClientData)NULL);
```

This creates a write trace on variable `x` in `interp`, which means that `WriteProc` is invoked whenever `x` is modified. The third argument to `Tcl_TraceVar` is an ORed combination of flag bits that select the operations to trace: `TCL_TRACE_READS` for reads, `TCL_TRACE_WRITES` for writes, and `TCL_TRACE_UNSETS` for

unsets. In addition, the flag TCL_GLOBAL_ONLY may be specified to force the variable name to be interpreted as global. Tcl_TraceVar and Tcl_TraceVar2 normally return TCL_OK. If an error occurs, they set an error result in the interpreter and return TCL_ERROR.
The library functions Tcl_UntraceVar and Tcl_UntraceVar2 remove variable traces. For example, the following call removes the trace set previously:

```
Tcl_Untrace(interp, "x", TCL_TRACE_WRITES,
    WriteProc,(ClientData)NULL);
```

Tcl_UntraceVar finds the specified variable; looks for a trace that matches the flags, trace function, and clientData specified by its arguments; and removes the trace if it exists. If no matching trace exists, Tcl_UntraceVar does nothing. Tcl_UntraceVar and Tcl_UntraceVar2 accept the same flags as Tcl_TraceVar.

34.7 Trace Callbacks

Trace callback functions such as WriteProc in the previous section must match the following prototype:

```
typedef char *Tcl_VarTraceProc(
    ClientData clientData, Tcl_Interp *interp,
    char *name1, char *name2, int flags);
```

The *clientData* and *interp* arguments are the same as the corresponding arguments passed to Tcl_TraceVar or Tcl_TraceVar2. *clientData* typically points to a structure containing information needed by the trace callback. *name1* and *name2* give the name of the variable in the same form as the arguments to Tcl_SetVar2. *flags* consist of an ORed combination of bits. Either TCL_TRACE_READS, TCL_TRACE_WRITES, or TCL_TRACE_UNSETS is set to indicate which operation triggered the trace, and TCL_GLOBAL_ONLY is set if the variable is a global variable that isn’t accessible from the current execution context; the trace callback must pass this flag back into functions like Tcl_GetVar2 if it wishes to access the variable. The bits TCL_TRACE_DESTROYED and TCL_INTERP_DESTROYED are set in special circumstances, as described in Section 34.10.
For read traces the callback is invoked just before the Tcl_GetVar or other variable read operations return the variable’s value; if the callback modifies the value of the variable, the modified value is returned. For write traces the callback is invoked after the variable’s value has been changed. The

callback can modify the variable to override the change, and this modified value is returned as the result of Tcl_SetVar or other variable write functions. For unset traces the callback is invoked after the variable has been unset, so the callback cannot access the variable. Unset callbacks can occur when a variable has been unset, when a Tcl procedure returns (thereby deleting all of its local variables), or when an interpreter is destroyed (thereby deleting all of the variables in the interpreter).

A trace callback function can invoke Tcl_GetVar and Tcl_SetVar or their equivalents to read and write the value of the traced variable. All traces on the variable are temporarily disabled while the callback executes, so calls to Tcl_GetVar, Tcl_SetVar, and the like do not trigger additional trace callbacks. As mentioned earlier, unset traces aren't invoked until after the variable has been deleted, so attempts to read the variable during unset callbacks fail. However, it is possible for an unset callback function to write to the variable, in which case a new variable is created.

This code sets a write trace that prints out the value of variable x each time it is modified:

```
Tcl_TraceVar(interp, "x", TCL_TRACE_WRITES, Print,
(ClientData)NULL);
...
char *Print(ClientData clientData, Tcl_Interp *interp,
            char *name1, char *name2, int flags) {
    char *value;
    value = Tcl_GetVar2(interp, name1, name2,
                   flags & TCL_GLOBAL_ONLY);
    if (value != NULL) {
        if (name2 == NULL) {
            printf("new value of %s is %s\n",
                   name1, value);
        } else {
            printf("new value of %s(%s) is %s\n",
                   name1, name2, value);
        }
    }
    return NULL;
}
```

Print must pass the TCL_GLOBAL_ONLY bit of its flags argument to Tcl_GetVar2 in order to make sure that the variable can be accessed properly. Tcl_GetVar2 should never return an error, but Print checks for one anyway and doesn't try to print the variable's value if an error occurs. In this example we don't use the Tcl_Obj variant of Tcl_GetVar, because Tcl_GetVar2 is simpler and more direct.

Trace callbacks normally return NULL values; a non-null value signals an error. In this case the return value must be a pointer to a static string containing an error message. The traced access aborts, and the error message is returned to whoever initiated that access. For example, if the access was invoked by a set command or $ substitution, a Tcl error results; if the access was invoked via Tcl_GetVar, it returns NULL and also sets an error result in the interpreter if the TCL_LEAVE_ERR_MSG flag was specified.
The following code uses a trace to make the variable zip read-only with the value 94114:

```
Tcl_Obj *numobj = Tcl_NewIntObj(94114);
Tcl_IncrRefCount(numobj);
Tcl_TraceVar(interp, "zip", TCL_TRACE_WRITES, Reject,
                (ClientData)numobj);
...
char *Reject(ClientData clientData, Tcl_Interp *interp,
                char *name1, char *name2, int flags) {
    Tcl_Obj *correct = (Tcl_Obj *) clientData;
    Tcl_SetVar2Ex(interp, name1, name2,
                correct, flags & TCL_GLOBAL_ONLY);
    return "variable is read-only";
}
```

The Reject function is a trace callback that is invoked whenever zip is written to. It returns an error message to abort the write access. Since zip has already been modified before Reject is invoked, Reject must undo the write by restoring the variable's correct value. The correct value is passed to the trace callback using its clientData argument. This implementation allows the same function to be used as the write callback for many different read-only variables; a different correct value can be passed to Reject for each variable.

34.8 Whole-Array Traces

You can create a trace on an entire array by specifying an array name to Tcl_TraceVar or Tcl_TraceVar2 without an element name. This creates a whole-array trace: the callback function is invoked whenever any of the specified operations is invoked on any element of the array. If the entire array is unset, the callback is invoked just once, with *name1* containing the array name and *name2* being NULL.

34.9 Multiple Traces

Multiple traces can exist for the same variable. When this happens, each of the relevant callbacks is invoked on each variable access. The callbacks are invoked in order from the one most recently created to the oldest. If there are both whole-array traces and individual element traces, the whole-array callbacks are invoked before element callbacks. If an error is returned by one of the callbacks, no subsequent callbacks are invoked.

34.10 Unset Callbacks

Unset callbacks are different from read and write callbacks in several ways. First of all, unset callbacks cannot return an error condition; they must always succeed. Second, two extra flags are defined for unset callbacks: `TCL_TRACE_DESTROYED` and `TCL_INTERP_DESTROYED`. When a variable is unset, all of its traces are deleted; unset traces on the variable are still invoked, but they are passed the `TCL_TRACE_DESTROYED` flag to indicate that the trace has now been deleted and won't be invoked anymore. If an array element is unset and there is a whole-array unset trace for the element's array, the unset trace is not deleted and the callback is invoked without the `TCL_TRACE_DESTROYED` flag being set.
If the `TCL_INTERP_DESTROYED` flag is set during an unset callback, it means that the interpreter containing the variable has been destroyed. In this case the callback must be careful not to use the interpreter at all, since the interpreter's state is in the process of being deleted. The callback should merely clean up its own internal data structures.

35. Creating New Tcl Commands

For everyone planning on interacting with Tcl's C API, whether through extensions to Tcl or by embedding Tcl, creating new Tcl commands is the primary means of integrating Tcl with your library or application.
Each Tcl command is implemented by a function written in C. When a Tcl command is invoked during script evaluation, Tcl calls this function to carry out the command. This chapter describes how to write command functions, how to register them in an interpreter, and how to manage the result of the command.

35.1 Functions Presented in This Chapter

- `Tcl_Command Tcl_CreateObjCommand(Tcl_Interp *`*`interp`*`,`
  `CONST char *`*`cmdName`*`, Tcl_ObjCmdProc `*`proc`*`,`
  `ClientData `*`clientData`*`,`
  `Tcl_CmdDeleteProc `*`deleteProc`*`)`

Creates a new command that is visible in Tcl as *`cmdName`*, which, when called, runs the function *`proc`*.

- `Tcl_WrongNumArgs(Tcl_Interp *`*`interp`*`, int `*`objc`*`,`
  `CONST Tcl_Obj *`*`objv`*`[], CONST char *`*`message`*`)`

Generates a standard error message and stores it in the result object of *`interp`*. The message includes the *`objc`* initial elements of *`objv`* plus *`message`*.

- `int Tcl_DeleteCommand(Tcl_Interp *`*`interp`*`,`
  `CONST char *`*`cmdName`*`)`

Deletes the command *`cmdName`*.

- `int Tcl_DeleteCommandFromToken(Tcl_Interp *`*`interp`*`,`
  `Tcl_Command `*`token`*`)`

Deletes the command referred to by *`token`*.

- `Tcl_SetObjResult(Tcl_Interp *`*`interp`*`, Tcl_Obj *`*`objPtr`*`)`

Sets *`objPtr`* as the result for *`interp`* and increments its reference count. This replaces any existing result and decrements the reference count for any old result object.

- `Tcl_SetResult(Tcl_Interp *`*`interp`*`, char *`*`result`*`,`
  `Tcl_FreeProc *`*`freeProc`*`)`

Arranges for *`result`* to be the result for the current Tcl command in *`interp`*, replacing any existing result. The *freeProc* argument specifies how to manage the storage for the result argument.

- `Tcl_AppendResult(Tcl_Interp *interp, char *result,`
  `char *result, ... , (char *) NULL)`
  `Tcl_AppendResultVA(Tcl_Interp *interp, va_list argList)`

Takes each of its *result* arguments and appends them in order to the current result associated with *interp*. `Tcl_AppendResultVA` is the same as `Tcl_AppendResult` except that instead of taking a variable number of arguments, it takes an argument list.

- `Tcl_AppendElement(Tcl_Interp *interp, char *element)`

Takes only a single *element* argument and appends that argument to the current result as a proper Tcl list element.

- `Tcl_ResetResult(Tcl_Interp *interp)`

Clears the result for *interp* and leaves the result in its normal empty initialized state. If the result is an object, its reference count is decremented.

- `int Tcl_GetCommandInfo(Tcl_Interp *interp,`
  `CONST char *cmdName, Tcl_CmdInfo infoPtr)`

Obtains information about the command *cmdName* and places it in *infoPtr*.

- `int Tcl_SetCommandInfo(Tcl_Interp *interp,`
  `CONST char *cmdName, Tcl_CmdInfo infoPtr)`

Modifies functions and client data associated with the command *cmdName*.

- `int Tcl_GetCommandInfoFromToken(Tcl_Command token,`
  `Tcl_CmdInfo infoPtr)`

Obtains information about the command referred to by *token* and places it in *infoPtr*.

- `int Tcl_SetCommandInfoFromToken(Tcl_Command token,`
  `Tcl_CmdInfo infoPtr)`

Modifies functions and client data associated with the command referred to by *token*.

- `CONST char *Tcl_GetCommandName(Tcl_Interp *interp,`
  `Tcl_Command token)`

Gets the name of the command referred to by *token*.

- `void Tcl_GetCommandFullName(Tcl_Interp *interp,`
  `Tcl_Command token, Tcl_Obj *objPtr)`

Produces the fully qualified name of a command from *token*. The name, including all namespace prefixes, is appended to the object specified by *objPtr*.

- `Tcl_Command Tcl_GetCommandFromObj(Tcl_Interp *interp,`
  `Tcl_Obj *objPtr)`

Returns a token for the command specified by the name in a `Tcl_Obj`. The command name is resolved relative to the current namespace. Returns `NULL` if the command is not found.

- `Tcl_Trace Tcl_CreateObjTrace(Tcl_Interp *`*`interp`*`,`
  `int` *`level`*`, int` *`flags`*`, Tcl_CmdObjTraceProc` *`objProc`*`,`
  `ClientData` *`clientData`*
  `Tcl_CmdObjTraceDeleteProc` *`deleteProc`*`)`

Creates a trace. The *`objProc`* function is called prior to the execution of every command in interpreter *`interp`*. Only commands at or below nesting level *`level`* are traced. Returns a trace that can be passed to `Tcl_DeleteTrace`.

- `Tcl_Trace Tcl_CreateTrace(Tcl_Interp *`*`interp`*`,`
  `int` *`level`*`, Tcl_CmdTraceProc` *`func`*`,`
  `ClientData` *`clientData`*`)`

Similar to `Tcl_CreateObjTrace`, but for command functions implemented using the older interface.

- `Tcl_DeleteTrace(Tcl_Interp *`*`interp`*`, Tcl_Trace` *`trace`*`)`

Deletes a trace.

35.2 Command Functions

The interface to an object-based command function is defined by the `Tcl_CmdProc` function prototype:

```
typedef int Tcl_ObjCmdProc(
    ClientData clientData, Tcl_Interp *interp,
    int objc, Tcl_Obj *CONST objv[]);
```

In other words, functions that implement new Tcl commands have this signature. This replaces the older-style string-based commands, which have the following function signature—included here only to aid in the understanding of old code that is in need of updating:

```
typedef int Tcl_CmdProc(
    ClientData clientData, Tcl_Interp *interp,
    int argc, char *argv[]);
```

A command function is invoked when its corresponding Tcl command is executed, and it is passed four arguments. The first, *`clientData`*, is discussed in Section 35.7. The second, *`interp`*, is the interpreter in which the command was executed. The third and fourth arguments are similar in meaning to the *`argc`* and *`argv`* arguments to a C `main` program: *`objc`* specifies the total number of arguments (words) to the Tcl command, and *`objv`* is an array of pointers to

the Tcl_Obj values of the words. Tcl processes all the special characters such as $ and [] before invoking command functions, so the values in *objv* are those left after any substitutions have been performed.
The name of the command is counted in *objc* (meaning the command cmd foo has an objc of 2) and is included as the first element of *objv*. *objv*[*objc*] is NULL.
A command function "returns" two values, just like Tcl_Eval and Tcl_EvalFile. The C function returns an integer completion code such as TCL_OK or TCL_ERROR and additionally sets a result in the Tcl interpreter with Tcl_SetObjResult. The Tcl_WrongNumArgs call is also commonly used to indicate an error condition caused by an incorrect number of arguments to the Tcl command.
Here is the command function for a new command called eq that compares its two arguments for string equality:

```
static int
EqCmd(ClientData clientData, Tcl_Interp *interp,
    int objc, Tcl_Obj *CONST objv[]) {
    char *arg1;
    char *arg2;
    Tcl_Obj *result;
    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv, "string1 string2");
        return TCL_ERROR;
    }
    arg1 = Tcl_GetString(objv[1]);
    arg2 = Tcl_GetString(objv[2]);
    if (strcmp(arg1, arg2) == 0) {
        result = Tcl_NewBooleanObj(1);
    } else {
        result = Tcl_NewBooleanObj(0);
    }
    Tcl_SetObjResult(interp, result);
    return TCL_OK;
}
```

EqCmd checks to see that it was called with exactly two arguments (i.e., the objc has a value of 3, which includes the command name), and if not, it uses Tcl_WrongNumArgs to set an error result and then returns TCL_ERROR. Otherwise, it fetches the string values of its two arguments, compares them, and creates a Boolean object that indicates whether they are indeed equal or not. It then returns TCL_OK to indicate that the command completed normally. EqCmd does not use its clientData argument, but this argument plays an important role in many other Tcl commands, as you will see shortly.

Note

> This command does not check the equivalence of two objects, only their string equality. Consider two numbers, 1 and 1.0—`EqCmd` will indicate that they are different.

A command function should treat the contents of the `objv` array as read-only. In general, it is not safe for a command function to modify these objects.

35.3 Registering Commands

In order for a command function to be invoked by Tcl, you must register it by calling `Tcl_CreateObjCommand`, which takes the form

```
Tcl_Command token Tcl_CreateObjCommand(
    Tcl_Interp *interp, char *commandName,
    Tcl_ObjCmdProc *proc, ClientData clientData,
    Tcl_CmdDeleteProc deleteProc);
```

This is the “magic” that associates the string in Tcl with the C function that implements it. Here is the simple program from Section 31.3, augmented with a call to `Tcl_CreateObjCommand`:

```
#include <stdio.h>
#include <tcl.h>

int main(int argc, char *argv[]) {
    Tcl_Interp *interp;
    char *result;
    int code;
    if (argc != 2) {
        fprintf(stderr, "Wrong # arguments: ");
        fprintf(stderr, "should be \"%s fileName\"\n",
                argv[0]);
        exit(1);
    }

    interp = Tcl_CreateInterp();
    Tcl_CreateObjCommand(interp, "eq", EqCmd,
                         (ClientData) NULL,
                         (Tcl_CmdDeleteProc *) NULL);
    code = Tcl_EvalFile(interp, argv[1]);
    result = Tcl_GetStringResult(interp);
    if (code != TCL_OK) {
        printf("Error was: %s\n", result);
        exit(1);
    }

    printf("Result was: %s\n", result);
    exit(0);
}
```

The first argument to `Tcl_CreateObjCommand` identifies the interpreter in which the command will be used. The second argument specifies the name for the command, and the third argument specifies its command function. The fourth and fifth arguments are discussed in Section 35.7; they can be specified as `NULL` for simple commands like this one. `Tcl_CreateObjCommand` creates a new command for `interp` named `eq`. If a command by that name already exists, it is silently deleted. Whenever `eq` is invoked in `interp`, Tcl calls `EqCmd` to carry out its function. The `Tcl_CreateObjCommand` function returns a `Tcl_Command` “token” that can be used to further manipulate and obtain information about the command. See Section 35.8 and Section 35.9 for further information.
If the code for `EqCmd` is included in the same file with the `main` function, it can be compiled and invoked just like the simple program in Section 31.3. Alternatively, `EqCmd` could be compiled into an extension and loaded, as described in Chapter 36. In either case, scripts are now able to use the new `eq` command:

```
   eq abc def
⇒ 0
   eq 1 1
⇒ 1
   set w .dlg
   set w2 .dlg.ok
   eq $w.ok $w2
⇒ 1
```

When processing scripts, Tcl carries out all of the command-line substitutions before calling the command function, so when `EqCmd` is called for the last `eq` command in the preceding code, both `objv[1]` and `objv[2]` contain objects with the string `.dlg.ok`.

`Tcl_CreateObjCommand` is usually called by applications during initialization to register application-specific commands, but new commands can be created at any time while an application is running—even by other commands. For example, the `proc` command creates a new command for each Tcl procedure that is defined, and Tk creates a widget command for each new widget. In Section 35.7 and Section 35.9 you will see examples where the command function for one command creates a new command.

Commands created by `Tcl_CreateObjCommand` are indistinguishable from Tcl's built-in commands. Each built-in command has a command function with the same form as `EqCmd`, and you can redefine a built-in command by calling `Tcl_CreateObjCommand` with the name of the command and a new command function.

35.4 The Result Protocol

The `EqCmd` function returns a result through a call to the `Tcl_SetObjResult` function. This is the most efficient way to set a result. In the example in the previous section, for instance, we create the type as a Boolean, so that if the `eq` command's result is read by another command that expects a Boolean as an argument, no translation is necessary. If we were to provide a string result, it would have to be created, then parsed back into a Boolean (an integer).

By setting an object as the result in the interpreter, Tcl automatically increments its reference count, so it's not necessary for you to do it explicitly.

35.5 `Tcl_AppendResult`

Another method of managing the result that is still useful, and reasonably current (it is used in Tcl internally), is `Tcl_AppendResult`, which makes it easy to build up results in pieces. It takes any number of strings as arguments and appends them to the interpreter's result in order. As the result grows in length, `Tcl_AppendResult` allocates new memory for it. `Tcl_AppendResult` may be called repeatedly to build up long results incrementally, and it does this efficiently, even if the result becomes very large (it allocates extra memory so that it doesn't have to copy the existing result into a larger area on each call). Here is an example from Tcl's source that sets an error message when it cannot open a file:

```
Tcl_AppendResult(interp, "couldn't read file \"",
            filename, "\": ",
            Tcl_PosixError(interp),
            (char *) NULL);
return TCL_ERROR;
```

The `NULL` argument in `Tcl_AppendResult` marks the end of the strings to append.
`Tcl_AppendElement` is similar to `Tcl_AppendResult` except that it adds only one string to the result at a time, and it appends it as a list element instead of a raw string. In modern Tcl applications, it is probably best to construct a list as an object and provide that as an argument to `Tcl_SetObjResult`.
If you set the result for an interpreter and then decide that you want to discard it (for example, an error has occurred and you want to replace the current result with an error message), you should call the function `Tcl_ResetResult`. It frees the current result and restores the interpreter's result to its initialized state. You can then store a new value in the result in any of the usual ways. You need not call `Tcl_ResetResult` if you're going to use `Tcl_SetObjResult` or `Tcl_SetResult` to store the new result, because these functions take care of freeing any existing result.

35.6 `Tcl_SetResult` and `interp->result`

Before the advent of Tcl objects, the `Tcl_SetResult` function was used, and in even older versions it was possible to set the result by manipulating fields of the interpreter structure directly.

Note

This is slow and error-prone, and may not be supported in future Tcl releases, as it has been deprecated for years. We describe it here to assist the reader who is dealing with older code—and, we hope, updating it.

The full definition of the `Tcl_Interp` structure, as visible outside the Tcl library, is

```
typedef struct Tcl_Interp {
    char *result;
    Tcl_FreeProc *freeProc;
    int errorLine;
} Tcl_Interp;
```

The first field, `result`, points to the interpreter's current result. It is possible to set it directly to a string, by doing, for instance,

```
interp->result = "Maximum temperature exceeded!";
```

Tcl also provides, by default, a small amount of storage space that can be written to directly. The exact quantity is defined by `TCL_RESULT_SIZE`, which is guaranteed to be at least 200. You may see this used for numeric results in code like the following:

```
sprintf(interp->result, "%d", argc);
```

It is also possible to allocate memory (via `malloc` or the like) and assign it to `interp->result`. It is at this point that the second field in the `interp` structure, `freeProc`, comes into play. If the result is set from dynamically allocated memory, `freeProc` is a function that can free the memory when it is no longer used. A `freeProc` must be defined with this signature:

typedef void Tcl_FreeProc(char **blockPtr*);

The function is invoked with a single argument containing the address stored in `interp->result`. In older code, `malloc` is used for dynamic allocation—as opposed to `ckalloc` or `Tcl_Alloc`, which should always be used in new code. Thus, `interp->freeProc` is usually set to the `free` function.
`Tcl_SetResult` is a step up from managing `interp->result` directly, yet it still requires a function to free memory once it is no longer in use. The first argument to `Tcl_SetResult` is an interpreter, the second argument is a string to use as the result, and the third argument gives additional information about

the string. `TCL_STATIC` means that the string is static, so `Tcl_SetResult` just stores its address into `interp->result`. A value of `TCL_VOLATILE` for the third argument means that the string is about to change (for example, it is stored in the function's stack frame), so a copy must be made for the result. `Tcl_SetResult` copies the string into the pre-allocated space if it fits; otherwise, it allocates new memory to use for the result and copies the string there (setting `interp->freeProc` appropriately). If the third argument is `TCL_DYNAMIC`, it means that the string was allocated with `malloc` and should become the property of Tcl: `Tcl_SetResult` sets `interp->freeProc` to `free` as described earlier. Finally, the third argument may be the address of a function suitable for use in `interp->freeProc`; in this case the string is dynamically allocated and Tcl eventually calls the specified function to free it.

Note

Once again: don't use either of these approaches for any new code you develop. If you come across them in existing code, take a few minutes to bring the code up to date.

35.7 `clientData` and Deletion Callbacks

The fourth and fifth arguments to `Tcl_CreateObjCommand`, `clientData` and `deleteProc`, were not discussed in Section 35.3, but they are useful when commands are associated with "objects" (not `Tcl_Obj`s, but objects as described in Section 30.3). The `clientData` argument is used to pass a one-word value to a command function (typically the address of the C data structure for an object). Tcl saves the `clientData` value and uses it as the first argument to the command function. The type `ClientData` is large enough to hold either an integer or a pointer value.

`clientData` values are used in conjunction with callback functions. A callback is a function whose address is passed to a Tcl library function and saved in a Tcl data structure. Later, at some significant time, the address is used to invoke the function ("call it back"). A command function is an example of a callback: Tcl associates the function's address with a Tcl command name and calls the function whenever the command is invoked. When a callback is specified in Tcl or Tk, a `clientData` argument is provided along with the

function's address, and the `clientData` value is passed to the callback as its first argument.
The `deleteProc` argument to `Tcl_CreateObjCommand` specifies a deletion callback. If its value isn't `NULL`, it is the address of a function for Tcl to call when the command is deleted. The function must match the following prototype:

```
typedef void Tcl_CmdDeleteProc(ClientData clientData);
```

The deletion callback takes a single argument, which is the *`clientData`* value specified when the command was created. Deletion callbacks are used for purposes such as freeing the object associated with a command.

```
static int
ObjectCmd(ClientData clientData,
          Tcl_Interp *interp, int objc,
          Tcl_Obj *CONST objv[]) {
    Counter *counterPtr = (Counter *)clientData;
    char *subcmd;
    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv,
                         "get | next");

        return TCL_ERROR;
    }
    subcmd = Tcl_GetString(objv[1]);
    if (strcmp(subcmd, "get") == 0) {
        Tcl_SetObjResult(interp,
            Tcl_NewIntObj(counterPtr->value));
    } else if (strcmp(subcmd, "next") == 0) {
        counterPtr->value ++;
    } else {
        Tcl_Obj *result = Tcl_NewStringObj("", 0);
        Tcl_AppendStringsToObj(result,
            "bad counter command\"",
            Tcl_GetString(objv[1]),
            "\": should be get or next",
            (char *)NULL);
        Tcl_SetObjResult(interp, result);
        return TCL ERROR;
    }
    return TCL_OK;
}

static int
CounterCmd(ClientData clientData, Tcl_Interp *interp,
        int objc, Tcl_Obj *CONST objv[]) {
    Counter *counterPtr;
    Tcl_Obj *countername;
    static int id = 0;
    if (objc != 1) {
        Tcl_WrongNumArgs(interp, 1, objv, NULL);
        return TCL_ERROR;
    }

    counterPtr = (Counter *)Tcl_Alloc(
                            sizeof(Counter));
    counterPtr->value = 0;
    countername = Tcl_NewStringObj("ctr", -1);
    Tcl_AppendObjToObj(countername,
                      Tcl_NewIntObj(id));
    id ++;
    Tcl_CreateObjCommand(interp,
                        Tcl_GetString(countername),
                        ObjectCmd,
                        (ClientData) counterPtr,
                        DeleteCounter);
    Tcl_SetObjResult(interp, countername);
    return TCL_OK;
}
```

These functions demonstrate how `clientData` and `deleteProc` can be used to implement counter objects. `CounterCmd` creates new counters, which are implemented by the `ObjectCmd` code. The application containing this code must register `CounterCmd` as a Tcl command using the following call:

```
Tcl_CreateObjCommand(interp, "counter", CounterCmd,,
                (ClientData) NULL,
                (Tcl_CmdDeleteProc *) NULL);
```

New counters can then be created by invoking the `counter` Tcl command; each invocation creates a new object and returns a name for that object:

```
  counter
⇒ ctr0
  counter
⇒ ctr1
```

`CounterCmd` is the command function for `counter`. It allocates a structure for the new counter and initializes its value to 0. Then it creates a name for the counter using the static variable `id`, arranges for that name to be returned as the command's result, and increments `id` so that the next new counter will get a different name.

This example uses the "object-oriented" style described in Section 30.3, where there is one command for each counter object. As part of creating a new counter, `CounterCmd` creates a new Tcl command named after the counter. It uses the address of the `Counter` structure as the `clientData` for the command and specifies `DeleteCounter` as the deletion callback for the new command.

Counters can be manipulated by invoking the commands named after them. Each counter supports two options to its command: `get`, which returns the current value of the counter, and `next`, which increments the counter's value. Once `ctr0` and `ctr1` are created, as shown earlier, the following Tcl commands can be invoked:

```
  ctr0 next; ctr0 next; ctr0 get
⇒ 2
  ctr1 get
⇒ 0
  ctr0 clear
⇒ bad counter command "clear": should be get or next
```

The function `ObjectCmd` implements the Tcl commands for all existing counters. It is passed a different `clientData` argument for each counter, which it casts back to a value of type `Counter *`. `ObjectCmd` checks `objv[1]` to see which command option was invoked. If it was `get`, it returns the counter's value as an integer object; if it was `next`, it increments the counter's value and does not set a result in the interpreter. If an unknown command was invoked, `ObjectCmd` calls `Tcl_AppendStringsToObj` together with `Tcl_SetObjResult` to create an error message.

To destroy a counter from Tcl you can delete its Tcl command; for example:

```
rename ctr0 {}
```

As part of deleting the command, Tcl invokes `DeleteProc`, which frees up the memory associated with the counter.

This object-oriented implementation of counter objects is similar to Tk's implementation of widgets: there is one Tcl command to create a new instance of a counter or widget, and one Tcl command for each existing counter or widget. A single command function implements all of the counter or widget commands for a particular type of object, receiving a `clientData` argument that identifies a specific counter or widget. A different mechanism is used to delete Tk widgets than to delete counters as shown earlier, but in both cases the command corresponding to the object is deleted at the same time as the object.

35.8 Deleting Commands

Tcl commands can be removed from an interpreter by calling `Tcl_DeleteCommand` (this function is the same for both object-based commands and old string-based commands). For example, the following statement deletes the `ctr0` command in the same way as the `rename` command used earlier:

```
Tcl_DeleteCommand(interp, "ctr0");
```

If the command has a deletion callback, it is invoked before the command is removed. Any command may be deleted, including built-in commands, application-specific commands, and Tcl procedures.
In the counter implementation, it would also be possible to provide a "delete" method in `ObjectCmd` that deletes the command, like so:

```
Tcl_DeleteCommand(interp, Tcl_GetString(objv[0]));
```

It is also possible to delete a command using the token returned by `Tcl_CreateObjCommand`:

```
Tcl_DeleteCommandFromToken(Tcl_Interp *interp,
                                        Tcl_Command token);
```

This works even if the command has been renamed.

35.9 Fetching and Setting Command Parameters

The Tcl library provides several functions for fetching and storing information about commands. The `Tcl_GetCommandInfo` and `Tcl_SetCommandInfo` calls work with the `Tcl_CmdInfo` structure, which has the following fields:

```
typedef struct Tcl_CmdInfo {
    int isNativeObjectProc;
    Tcl_ObjCmdProc *objProc;
    ClientData objClientData;
    Tcl_CmdProc *proc;
    ClientData clientData;
    Tcl_CmdDeleteProc *deleteProc;
    ClientData deleteData;
    Tcl_Namespace *namespacePtr;
} Tcl_CmdInfo;
```

The fields have the following meanings:

- `isNativeObjectProc`

Boolean value equal to `1` if the command was created with the "object API" `Tcl_CreateObjCommand`, or `0` if it is not object-based and was created with the "string API" `Tcl_CreateCommand` call. If this field is true, it is likely to be faster to call `objProc` instead of `proc`.

- `objProc`

A pointer to the function that implements the command.

- `objClientData`

Client data for the object API version of the command.

- `proc`

If this command was created with `Tcl_CreateCommand`, this points to the function that implements the command. If the command was brought into existance by `Tcl_CreateObjCommand`, this is simply a function that exists to provide compatibility with the older string API implementation.

- `clientData`

Client data for the string API command.

- `deleteProc`

This function is the same for both object and string API-derived commands and points to the function called when the command is deleted.

- `deleteData`

The `clientData` value passed to `deleteProc`.

- `namespacePtr`

A pointer to the namespace that the command is in.
The following command function clones a command by fetching information about one command using `Tcl_GetCommandInfo` and creating the new command:

```
static int
CloneCmd(ClientData clientData, Tcl_Interp *interp,
        int objc, Tcl_Obj *CONST objv[]) {
    char *oldCmdName;
    char *newCmdName;
    Tcl_CmdInfo cmdinfo;
    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv,
                        "oldCmdName newCmdName");
        return TCL_ERROR;
    }
    oldCmdName = Tcl_GetString(objv[1]);
    newCmdName = Tcl_GetString(objv[2]);
    if (Tcl_GetCommandInfo(
        interp, oldCmdName, &cmdinfo) == 0) {
        Tcl_AppendResult(interp,
                        "Command not found: ",
                        oldCmdName, NULL);
        return TCL_ERROR;
    }

    Tcl_CreateObjCommand(interp, newCmdName,
                        cmdinfo.objProc,
                        cmdinfo.objClientData,
                        cmdinfo.deleteProc);
    return TCL_OK;
}
```

In this code, `oldCmdName` is the command to be cloned, for example, `expr`. The function raises an error if a valid name is not provided and `Tcl_GetCommandInfo` returns `0`. `newCmdName` is the name of a new command to create with the same functionality as the existing command.
The same result could be achieved by replacing the `Tcl_CreateObjCommand` with the following code:

```
Tcl_CreateObjCommand(interp, newCmdName,
                    (Tcl_ObjCmdProc *)NULL,
                    (ClientData)NULL,
                    (Tcl_CmdDeleteProc *)NULL);
Tcl_SetCommandInfo(interp, newCmdName, &cmdinfo);
```

This creates an “empty” command and then sets it to use the data contained in the `cmdinfo` structure.
In addition to looking up commands by their names, it’s also possible to look them up via *tokens*. There are two ways to obtain a token for a given command. If you want to be certain you always have a token for a given command, you should store the token returned by `Tcl_CreateObjCommand`. This

always points to that command, even if it is renamed. You can also retrieve the token associated with a command name with the `Tcl_GetCommandFromObj` function. As an example, this code gets the information for the `puts` command:

```
Tcl_Command token;
token = Tcl_GetCommandFromObj(
        interp, Tcl_NewStringObj("puts", -1));
```

The commands that use `Tcl_Command` tokens to fetch and set command parameters are essentially the same as those described previously and are listed in Section 35.1.

35.10 How Tcl Procedures Work

All Tcl commands, including those in the Tcl core such as `set`, `if`, and `proc`, use the mechanism described in this chapter.

Note

In reality, the most common commands are bytecode-compiled, but that is an internal optimization not covered in this book.

For example, consider the creation and execution of a Tcl procedure:

```
  proc inc {x} { expr { $x + 1 } }
  inc 23
⇒ 24
```

When the `proc` command is evaluated, Tcl invokes its command function. The `proc` command function is part of the Tcl library, but it has the same arguments and results as described in Section 35.2. The `proc` command function allocates a new data structure to describe `inc`, including information about `inc`’s arguments and body. Then the `proc` command function invokes `Tcl_CreateObjCommand` to create the Tcl command for `inc`. It specifies a special function called `TclObjInterpProc` (which is an internal part of the Tcl library) as the command function for `inc`, and it uses the address of the new data structure as the `clientData` for the command. Then the `proc` command function returns.

When the `inc` procedure is evaluated, Tcl invokes `TclObjInterpProc` as the command function for the command. `TclObjInterpProc` uses its `clientData` argument to gain access to the data structure for `inc` and then byte-compiles the body of the procedure if necessary. Subsequently, `TclCompEvalObj` (another function internal to the Tcl library) evaluates `inc`'s body. Before calling `TclCompEvalObj`, `TclObjInterpProc` creates a new variable scope for the procedure, uses its `objv` argument to retrieve the first argument to the `inc` command, and assigns this value to the `x` variable in the new scope. When the `TclCompEvalObj` call completes, `TclObjInterpProc` destroys the procedure's variable scope and returns the completion code from `TclCompEvalObj`. All Tcl procedures have `TclObjInterpProc` as their command function. However, each Tcl procedure has a different `clientData`, which refers to the unique structure describing that procedure.

35.11 Command Traces

In addition to being able to trace variable access, as discussed in Section 34.6, it is also possible to trace command execution in Tcl. The basic method of doing this is via the `Tcl_CreateObjTrace` function, which creates a trace function with the following signature:

```
typedef int Tcl_CmdObjTraceProc(
    ClientData clientData,
    Tcl_Interp* interp,
    int level,
    CONST char* command,
    Tcl_Command commandToken,
    int objc,
    Tcl_Obj *CONST objv[] );
```

This is useful for creating low-level Tcl debuggers, because at each step of the program you can examine the command that's about to be executed, the program's state, and so on. This is a very powerful feature, although do keep in mind that because using it turns off Tcl's bytecode compiler, it notably slows down execution of Tcl code.

36. Extensions

Chapter 30 compared the approach of embedding Tcl into an existing application versus extending Tcl with custom commands. It's usually easier, and faster in terms of programmer time, to code a few extensions in C for an application written in Tcl, rather than write the application in C and call Tcl scripts from it. Writing an extension minimizes the amount of C programming compared to Tcl programming, making for better use of programmer time.
This chapter explains how to write a Tcl extension in C.

36.1 Functions Presented in This Chapter

- `CONST char *Tcl_PkgPresent(Tcl_Interp *`*`interp`*`,`
  `CONST char *`*`name`*`, CONST char *`*`version`*`, int `*`exact`*`)`
  `CONST char *Tcl_PkgRequire(Tcl_Interp *`*`interp`*`,`
  `CONST char *`*`name`*`, CONST char *`*`version`*`, int `*`exact`*`)`

Equivalent to the `package present` and `package require` commands, respectively. Returns a pointer to the version string for the version of the package that is provided in the interpreter (which may be different from *`version`*); if an error occurs, it returns `NULL` and leaves an error message in the interpreter's result.

- `int Tcl_PkgProvide(Tcl_Interp *`*`interp`*`,`
  `CONST char *`*`name`*`, CONST char *`*`version`*`)`

Equivalent to the `package provide` command. Returns `TCL_OK` if it completes successfully; if an error occurs, it returns `TCL_ERROR` and leaves an error message in the interpreter's result.

- `CONST char *Tcl_PkgRequireEx(Tcl_Interp *`*`interp`*`,`
  `CONST char *`*`name`*`, CONST char *`*`version`*`, int `*`exact`*`,`
  `ClientData *`*`clientDataPtr`*`)`
  `CONST char *Tcl_PkgPresentEx(Tcl_Interp *`*`interp`*`,`
  `CONST char *`*`name`*`, CONST char *`*`version`*`, int `*`exact`*`,`
  `ClientData *`*`clientDataPtr`*`)`
  `int Tcl_PkgProvideEx(Tcl_Interp *`*`interp`*`,`
  `CONST char *`*`name`*`, CONST char *`*`version`*
  `ClientData `*`clientData`*`)`

These calls are identical to the previous versions, but allow the setting and retrieving of the client data associated with the package.

- `CONST char *Tcl_InitStubs(Tcl_Interp *interp`
  `CONST char *version, int exact)`

This function must be the first one called from the Tcl library in an extension compiled with Tcl stubs. It attempts to initialize the stub table pointers.

36.2 The `Init` Function

In practical terms, an extension is a shared library that is loaded into Tcl at runtime with the `load` command, which is commonly wrapped in some Tcl code in order to make it possible to load C extensions in the same way as pure Tcl extensions—with a command like `package require XYZ`. Being shared objects, extensions are recognizable by the file name extension on the platform in question. Extension files end in `.dll` on Windows and `.so` on most Unix systems.

Tcl extensions are usually based around one or more new Tcl commands that provide functionality not available in the Tcl core. Proper extensions are written as packages, just as if they were packages of Tcl scripts.

The entry point for any extension is its `Init` function, which takes the form

```
int Myextension_Init(Tcl_Interp *interp) {
    ...
}
```

The function name, `Myextension_Init` in this case, must take the form *`Packagename`*`_Init`. *`Packagename`* corresponds to the extension's file name: an extension named `libPackagename.so` must use `Packagename_Init` as its `Init` function, and an extension with the `Init` function `Foopackage_Init` must be named `libFoopackage.so`. This is so that Tcl can determine the `Init` function's name and then call it, knowing only the shared library file name. In reality, it is possible to get around this requirement, but it's a good idea to stick to the convention, as it makes things easier.

The `Init` function is called when the extension is loaded and is run only once, even if the file is loaded multiple times. This function is where all setup takes place—commands are registered, variables are created, and data structures are initialized. For example, if you want to create a hash table to track a particular kind of object, create it in the `Init` function.

This function must return `TCL_OK` to indicate success, or `TCL_ERROR` if something went awry.

36.3 Packages

Similar to the pure Tcl package system covered in Chapter 14, compiled extensions offer a way to `provide` a package, based on the `Tcl_PkgProvide` function:

```
int Tcl_PkgProvide(Tcl_interp *interp, char *name,
        char *version);
```

The *`interp`* argument is an interpreter, *`name`* is the name of the package, and *`version`* is a version number string, such as `1.2` or `5.2.7`. As with pure Tcl packages, the name and version are used when the `package require` command is called in a script. Try to pick a unique name for your package, so as not to conflict with other packages.

Of course, in complex systems, your extension may require other extensions to function. It is possible to require other packages via the `Tcl_PkgRequire` function:

```
Tcl_PkgRequire(Tcl_Interp *interp, char *name,
        char *version, int exact);
```

The *`name`* is the name of a package to require. The *`version`* number is the minimum version of the package that will work. If the *`exact`* flag is not `0`, the version number must match exactly; otherwise, any number that is equal to or greater than the *`version`* provided here satisfies the dependency.

If you need to associate data with your package, in the form of a `ClientData` pointer, Tcl provides the `Tcl_PkgProvideEx` and `Tcl_PkgRequireEx` functions. Aside from taking a `ClientData` argument, they are identical to the function calls described previously.

When creating the code to build and install your Tcl extension (see Chapter 47), you also need to provide a `pkgIndex.tcl` file similar to the following:

```
package ifneeded counter 0.1 \
    [list load [file join $dir \
        libcounter[info sharedlibextension]]]
```

This is what actually carries out the `load` when a script asks for the package with `package require counter`. Automatic generation of `pkgIndex.tcl` files is explained in further detail in Chapter 14.

36.4 Namespaces

Tcl namespaces aid the programmer in modularizing and encapsulating code. Tcl namespaces are discussed from the scripting level in Chapter 10. In current versions of Tcl there is no C API to the namespace features, but there are some work-arounds to create new namespaces from C.
Creating a command in a namespace automatically creates the namespace as well if it does not already exist:

```
Tcl_CreateObjCommand(interp, "foo::bar", BarCmd,
        (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);
```

You can also create it with a quick `Tcl_Eval` call:

```
char *nsscript = "namespace eval ::foo {}";
Tcl_Eval(interp, nsscript);
```

If you are just providing a command or two, you probably don't need to create a namespace, but if you have many commands, and especially if you are creating lots of variables, consider using a namespace in order to spare other programmers the chore of looking after too many commands in the global namespace.

36.5 Tcl Stubs

Tcl provides a mechanism to ensure that an extension functions with different versions of Tcl without having to be compiled or linked against each one. This greatly increases compatibility between different releases of extensions and Tcl versions, which are not required to move in lockstep with one another. It does this by performing the symbol table lookups itself instead of using the normal shared library mechanism. This double indirection is what lets us detach the extension in question from a particular version of the Tcl library. The stubs mechanism is invoked in a Tcl extension via the function `Tcl_InitStubs`:

```
Tcl_InitStubs(Tcl_Interp *interp,
              char *version, int exact);
```

The first argument specifies the interpreter in which the extension will be loaded. The *`version`* string argument specifies the minimum version of Tcl required for the extension (the major number of the version must match that of the Tcl interpreter). For example, `8.3` indicates that the extension needs

Tcl version 8.3 or later, whereas `8` indicates that any version 8 Tcl interpreter is sufficient. If you want to use a specific version of the Tcl library, set the *`exact`* flag to `1`, requiring that the stubs match one, and only one, version of Tcl—although at this point it might make more sense to link directly and not use the stubs mechanism.

`Tcl_InitStubs` must be the first call in a Tcl extension that uses stubs; to access other functions in the Tcl library, the stubs must be loaded.

In order to write robust code, it's usually a good idea to use a `#ifdef` to isolate the `Tcl_InitStubs` call, in case someone compiles the extension without the stubs mechanism:

```
#ifdef USE_TCL_STUBS
    if (Tcl_InitStubs(interp, "8", 0) == NULL) {
        return TCL_ERROR;
    }
#endif
```

Thus, the use of stubs is determined by a compile-time flag, which is covered in greater depth in Chapter 47.

36.6 The `ifconfig` Extension

The following source code implements an extension called `ifconfig` for querying the status of network interfaces on a Linux system:

```
/*
  Copyright 2004, David N. Welton <davidw@dedasys.com>

  This code may be used, modified and redistributed under the
  same terms as Tcl itself.
*/

#include <stdlib.h>
#include <strings.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <tcl.h>


#define IFC_BUFLEN 8192

/*
 *-----------------------------------------------------------------
 *
 *  ListInterfaces --
 *
 *  List available interfaces.
 *
 * Results:
 *  A list of interfaces.
 *
 * Side Effects:
 *  None.
 *
 *-----------------------------------------------------------------
 */

static int ListInterfaces(Tcl_Interp *interp) {
    struct ifconf if_request;
    char *ifc_buffer;
    int i, j;
    int fd;
    Tcl_Obj *iflist;
    Tcl_Obj *ifname = NULL;

    /* Allocate space for the interface request. */
    ifc_buffer = (char *)malloc(IFC_BUFLEN);

    bzero(ifc_buffer, IFC_BUFLEN);
    bzero(&if_request, sizeof(if_request));
    /* We need a socket file descriptor to work with, even if it
     * isn't connected to anything. */
    fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);

    if_request.ifc_buf = ifc_buffer;
    if_request.ifc_len = IFC_BUFLEN;

    /* Fetch information from the kernel. */
    ioctl(fd, SIOCGIFCONF, &if_request);
    iflist = Tcl_NewObj();
    /* Loop through interfaces. */
    for (i = 0,  j = 0; i < if_request.ifc_len;
         i += sizeof(struct ifreq), j++) {
       /* Fetch interface name, create a new string object. */
       ifname = Tcl_NewStringObj(if_request.ifc_req[j].ifr_name,
                                 -1);

       /* Add it to the list. */
       Tcl_ListObjAppendElement(interp, iflist, ifname);
    }
    /* Use the list as the result. */
    Tcl_SetObjResult(interp, iflist);

    /* Clean up other resources. */
    close(fd);
    free(ifc_buffer);

    return TCL_OK;
}

/*
 *----------------------------------------------------------------


 *
 * GetInterface --
 *
 *  Fetch information about a particular interface in the form
 *  of a key/value list suitable for use as an array or dict.
 *
 * Results:
 *  A list of keys and their values.
 *
 * Side Effects:
 *  None.
 *
 *----------------------------------------------------------------
 */

static int GetInterface(Tcl_Interp *interp,
           int objc,
           Tcl_Obj *CONST objv[]) {
    int fd;

    struct ifreq ifreq;
    unsigned char *hw;
    struct sockaddr_in *sin;
    char *itf;
char hwaddr[40];
Tcl_Obj *addr;
Tcl_Obj *braddr;
Tcl_Obj *netmask;
Tcl_Obj *result;

if (objc != 3) {
    Tcl_WrongNumArgs(interp, 2, objv, "interface");
    return TCL_ERROR;
}
itf = Tcl_GetString(objv[2]);

fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_IP);

strcpy(ifreq.ifr_name, itf);

/* hardware address */
ioctl(fd, SIOCGIFHWADDR, &ifreq);
hw = ifreq.ifr_hwaddr.sa_data;
sprintf(hwaddr, "%02x:%02x:%02x:%02x:%02x:%02x",
        *hw, *(hw + 1), *(hw + 2), *(hw + 3),
        *(hw + 4), *(hw + 5));



result = Tcl_NewObj();
Tcl_ListObjAppendElement(interp, result,
             Tcl_NewStringObj("hwaddr", -1));
Tcl_ListObjAppendElement(interp, result,
             Tcl_NewStringObj(hwaddr, -1));

/* address */
if (ioctl(fd, SIOCGIFADDR, &ifreq) == 0) {
    sin = (struct sockaddr_in *)&ifreq.ifr_broadaddr;
    addr = Tcl_NewStringObj(inet_ntoa(sin->sin_addr), -1);

   /* broadcast */
   ioctl(fd, SIOCGIFBRDADDR, &ifreq);
   sin = (struct sockaddr_in *)&ifreq.ifr_broadaddr;
   braddr = Tcl_NewStringObj(inet_ntoa(sin->sin_addr), -1);

   /* netmask */
   ioctl(fd, SIOCGIFNETMASK, &ifreq);
   sin = (struct sockaddr_in *)&ifreq.ifr_broadaddr;
   netmask = Tcl_NewStringObj(inet_ntoa(sin->sin_addr), -1);
} else {
   addr = Tcl_NewObj();
   braddr = Tcl_NewObj();
   netmask = Tcl_NewObj();
}
```

```
    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewStringObj("addr", -1));
    Tcl_ListObjAppendElement(interp, result, addr);
    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewStringObj("braddr", -1));
    Tcl_ListObjAppendElement(interp, result, braddr);
    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewStringObj("netmask", -1));
    Tcl_ListObjAppendElement(interp, result, netmask);

    /* flags */
    ioctl(fd, SIOCGIFFLAGS, &ifreq);

    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewStringObj("up", -1));
    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewBooleanObj(
                (ifreq.ifr_flags & IFF_UP)));

    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewStringObj("running", -1));
    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewBooleanObj(
                (ifreq.ifr_flags & IFF_RUNNING)));



    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewStringObj("noarp", -1));
    Tcl_ListObjAppendElement(interp, result,
                Tcl_NewBooleanObj(
                (ifreq.ifr_flags & IFF_NOARP)));


    close(fd);

    Tcl_SetObjResult(interp, result);
    return TCL_OK;
}

/*
 *-------------------------------------------------------------
 *
 * Ifconfig --
 *
 *  The main function, which calls secondary functions depending
 *  on the "subcommand".
 *
 * Results:
 *  Depends on the function requested.
 *
 * Side Effects:
 *  None.
 *
 *----------------------------------------------------------------
 */

static int Ifconfig(ClientData clientData,
            Tcl_Interp *interp,
            int objc,
            Tcl_Obj *CONST objv[]) {
    char *subcmd;
    if (objc < 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "list | get | set");
        return TCL_ERROR;
    }

    subcmd = Tcl_GetString(objv[1]);

    if (strcmp(subcmd, "list") == 0) {
        return ListInterfaces(interp);
    } else if (strcmp(subcmd, "get") == 0) {
        return GetInterface(interp, objc, objv);
    } else {
        return TCL_ERROR;


    }
    return TCL_OK;
}

/*
 *----------------------------------------------------------------
 *
 * Tclifconfig_Init --
 *
 *  Initializes the package, creating the ifconfig command.
 *
 * Results:
 *  None.
 *
 * Side Effects:
 *  Creates ifconfig command and package.
 *
 *----------------------------------------------------------------
 */
```

```
int Tclifconfig_Init(Tcl_Interp *interp) {

#ifdef USE_TCL_STUBS
    if (Tcl_InitStubs(interp, "8", 0) == NULL) {
    return TCL_ERROR;
    }
#endif

    /* Create the command. */
    Tcl_CreateObjCommand(interp, "ifconfig", Ifconfig,
            (ClientData) NULL, (Tcl_CmdDeleteProc *) NULL);

    if ( Tcl_PkgProvide(interp,
            "ifconfig",
            "0.1") != TCL_OK ) {
        return TCL_ERROR;
    }
    return TCL_OK;
}
```

Consider the `Init` function. The compiled name of this library in Linux is `libtclifconfig.so`, so `tclifconfig` corresponds to `Tclifconfig` in the `Tclifconfig_Init` function. When Tcl loads this library, it looks for this function based on the name of the shared library. At that point, depending on whether we compiled with the `USE_TCL_STUBS` flag, we may or may not load the stub table.
The next order of business is to create the `ifconfig` command. We associate it with the `Ifconfig` function using `Tcl_CreateObjCommand`, which also registers the `ifconfig` command for use from within Tcl scripts.
The `Tcl_PkgProvide` call that comes next registers the extension as the `ifconfig` package—not to be confused with the command—and sets the version to 0.1.
The `Init` function must return a `TCL_OK`, or else the extension fails to load.
After compiling the extension, we can use the new `ifconfig` command to gather information about network interfaces on a Linux system:

```
  load ./libtclifconfig.so
  ifconfig list
⇒ lo eth0 eth1
  ifconfig get eth1
⇒ hwaddr 00:02:2d:1f:41:46 addr 10.0.0.10 braddr 10.255.255.255
  netmask 255.255.255.0 up 1 running 64
  noarp 0
  array set eth1 [ifconfig get eth1]
  set eth1(hwaddr)
⇒ 00:02:2d:1f:41:46
```

All things considered, the source code, with comments, weighs in at less

than 300 lines of C and returns data in a form that we can use very easily in Tcl.
By way of comparison, consider another way of performing the same task, parsing the output of the `/sbin/ifconfig` command. In that case you must be absolutely sure that the output format will not change from version to version or among different platforms. If it does change, your script may have to deal with mistaken information. Our Tcl/C extension requires some cross-platform dexterity (`ifdef`s) of its own, but on the other hand, it’s very fast, and there is no parsing involved—the data is immediately ready to use with other Tcl commands.
To package this code for distribution, it would be a good idea to create a `pkgIndex.tcl` file along these lines:

```
if {[catch {package present ifconfig 1.0}]} { return }
    package ifneeded ifconfig 1.0 [list load \
        [file join /usr/lib libifconfig1.0.so.0] Tk]
```

37. Embedding Tcl

Embedding is the practice of using Tcl from within an existing application written in C. This chapter explains how to add Tcl to your application, as well as how to create new `tclsh`-style applications.

37.1 Functions Presented in This Chapter

- `void Tcl_FindExecutable(argv[0])`

Computes the path of the executable, which is needed for several mechanisms internal to Tcl.

- `CONST char *Tcl_GetNameOfExecutable()`

Returns a path to the full path name of the application.

- `int Tcl_Init(Tcl_Interp *`*`interp`*`)`

Carries out various Tcl initialization tasks, such as sourcing Tcl's own `init.tcl`.

- `int Tcl_AppInit(Tcl_Interp *`*`interp`*`)`

User-supplied hook procedure used in the creation of new `tclsh`-style programs.

37.2 Adding Tcl to an Application

If you're creating a new application from scratch, it may be worth considering writing the application in Tcl, but for many reasons this might not be an option. Embedding Tcl in existing applications is easy—with a little bit of code, it's possible to make an application far more powerful by giving it a Tcl interpreter. In the extension model we saw earlier, most of the code is written in Tcl, and extensions written in C are added as needed to provide additional functionality or speed. In the "embedding" model, Tcl is just another library that's linked in to the main application to provide a service, in this case, the evaluation of Tcl scripts.
For example, you could take a web server and add Tcl to it, allowing users to create dynamic web pages in Tcl. This is the approach that AOLserver and the various Apache Tcl projects have taken. The open-source

PostgreSQL database lets you use Tcl to write functions and trigger procedures. There are countless other examples of large, complex programs that have given users the ability to script new and interesting tools with Tcl. A lot of value can be added to a program if it has a powerful scripting interface. Your users will be able to create and share code—perhaps they will even discover innovative ways of hooking the product to other systems through Tcl.
Chapter 31 contains an extremely simple application that evaluates a Tcl script contained in a file. Embedding Tcl doesn't require much more work.
You need to choose a point in the life cycle of your application where you want to initialize Tcl. For instance, you may wish to use Tcl as a configuration language—you could start Tcl, and then use it to read in a file that sets options for your system, such as

```
set port 80
set hostname "www.tcl.tk"
set errorlog /var/log/myserver/error.log
```

Alternatively, you could create some commands so that you don't even have to treat these configuration options like variables:

```
port 80
hostname "www.tcl.tk"
errorlog /var/log/server/error.log
```

The command itself sets the variable internal to your application when it is called. If you already have a configuration system, you may wish to delay Tcl startup until you can set up the Tcl environment to reflect the information about the system it's running in; at that point you might set some Tcl variables and vary behavior depending on how the system has been configured.

37.3 Initialize Tcl

To evaluate Tcl code, you need to call the following setup functions:

```
...
Tcl_Interp *interp;
Tcl_FindExecutable(argv[0]);
interp = Tcl_CreateInterp();

if (interp == NULL) {
    fprintf(stderr, "Tcl Interp creation failed, exiting\n");
    exit(1);
}

if (Tcl_Init(interp) == TCL_ERROR) {
    fprintf(stderr, "Tcl Init failed, exiting\n");
    exit(1);
}
...
```

You should call `Tcl_FindExecutable` before any other function of the Tcl library, passing it `argv[0]` as its argument, to help the Tcl runtime to initialize itself. Then you can create an interpreter with `Tcl_CreateInterp`, as described in Chapter 31. If either the interpreter creation or the next call, `Tcl_Init`, should fail, your system has a problem, and you may wish to abort the application, as you will not be able to use Tcl inside it. `Tcl_Init` is the function responsible for finding and evaluating the `init.tcl` script that is distributed as a part of Tcl. Failure in any of these functions may indicate that you have not installed Tcl correctly.

The Tcl library is now ready to evaluate Tcl scripts. In the same location in your C code you should probably take care of any other initialization that you need, such as the creation of commands, variables, channels, and so on. If you are using multiple interpreters, you may want to initialize them first and then, in each interpreter that needs it, perform the setup tasks, as it is currently not possible to copy an interpreter's state (commands, variables, and so on).

The second decision you have to make, after having placed the Tcl initialization code in your application, is when to evaluate Tcl scripts. For some applications it's obvious when this needs to happen. In a web server that uses Tcl to dynamically generate pages, a Tcl script is evaluated when the user requests some resource (usually a web page). If the user has requested, say, `index.tcl`, the web server is responsible for executing that Tcl code and sending the HTML results back to the user. Other applications may need to add more complex hooks, such as working with Tcl's event loop (described in Chapter 43). However, at some point your application will call one of the functions in the `Tcl_Eval` family to process Tcl code.

37.4 Creating New Tcl Shells

Before Tcl gained the ability to load shared library extensions, a popular way of enhancing the language's functionality was to create new versions of the Tcl shell (`tclsh`) with the desired functionality added in. This is a special case of "embedding" where the only functionality of the C "application" is to set up and run Tcl code. The drawback to this approach is that you may want to use more than one extension, and at this point you would need to either integrate and compile both extensions or load one of them. Loading extensions dynamically is a far more flexible approach.

There may be situations where old code is involved or where you wish to have one statically linked executable with no libraries to load. In those cases, creating your own Tcl shell may be the best approach, although even here solutions like Starkits (described in Chapter 14) may be preferable.

You should also be aware of the differences between the Tcl library and the `tclsh` program. `tclsh` provides several variables that are not set up by the Tcl library, including `argc`, `argv`, `argv0`, and `tcl_interactive`, which are covered in Chapter 3. Additionally, `tclsh` sources a "dot-file," `.tclshrc`. If you wish to create a shell of your own, you most likely want to copy this behavior, so that your shell behaves like `tclsh`.

The means that Tcl provides to accomplish this is the `Tcl_AppInit` function. `Tcl_AppInit` is a function that performs application-specific initialization, such as creating new commands. The `main` functions for `tclsh` and `wish` invoke `Tcl_AppInit` after they have performed all of their own initializations, but before they start evaluating Tcl scripts. The overall idea is for the function `main` to carry out functions that are the same in all `tclsh`-like (or `wish`-like) applications, while `Tcl_AppInit` carries out the functions that may differ from application to application.

The version of `Tcl_AppInit` used by `tclsh` looks like this:

```
int Tcl_AppInit(Tcl_Interp *interp) {
    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    tcl_RcFileName = "~/.tclshrc";
    return TCL_OK;
}
```

`Tcl_AppInit` is invoked with a single argument consisting of the Tcl interpreter for the application. If `Tcl_AppInit` completes normally, it returns `TCL_OK`, and if it encounters an error, it returns `TCL_ERROR` and sets a result in the interpreter (in this case the application prints the error message and exits). This particular version of `Tcl_AppInit` does two things. First, it calls the function `Tcl_Init`, which performs additional Tcl initialization (it sources a script from the Tcl library to define the `unknown` procedure and set up the autoloading mechanism, described in Chapter 14). The argument and result for `Tcl_Init` are the same as for `Tcl_AppInit`. The second thing `Tcl_AppInit` does is to set the global variable `tcl_RcFileName`, which gives the name of a user-specific startup script. This variable is initially set to `NULL` by the `main` function in the default `tclsh` and `wish`; if `Tcl_AppInit` modifies `tcl_RcFileName`, and if the application is running interactively (rather than from a script file), `main` sources the file named by `tcl_RcFileName`.

To create a new `tclsh`-style shell, all you have to do is write your own `Tcl_AppInit` function and compile it with the Tcl library. For example, you can regenerate the `tclsh` application by making a copy of the file `unix/tclAppInit.c` from the Tcl source directory, which contains the preceding code for `Tcl_AppInit`. Then compile this file and link it with the Tcl library as described in Chapter 46:

```
cc tclAppInit.c -ltcl -lm -o mytclsh
```

The resulting application is identical to the system version of `tclsh`: it supports interactive input, script files, and all of the other features of `tclsh`.

If you wanted to include the `counter` package presented in Chapter 35 in your application, you might write the following `Tcl_AppInit`:

```
int Tcl_AppInit(Tcl_Interp *interp) {
    if (Tcl_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    if (Counter_Init(interp) == TCL_ERROR) {
        return TCL_ERROR;
    }
    tcl_RcFileName = "~/.tclshrc";
    return TCL_OK;
}
```

You would then compile `counter.c` as part of the application in addition to the file containing the `Tcl_AppInit` function described previously.

38. Exceptions

Many Tcl commands, such as `if` and `while`, have arguments that are Tcl scripts. The C functions for these commands evaluate them recursively to evaluate the scripts. If `Tcl_EvalObjEx` or any of its related evaluation functions returns a completion code other than `TCL_OK`, an exception is said to have occurred. Exceptions include `TCL_ERROR`, which was described in Chapter 31, plus several others that have not been mentioned yet. This chapter introduces the full set of exceptions and describes how to unwind nested evaluations and leave useful information in the `errorInfo` and `errorCode` variables.

38.1 Functions Presented in This Chapter

- `Tcl_AddObjErrorInfo(Tcl_Interp *`*`interp`*`,`
  `char *`*`message`*`, int `*`length`*`)`

Adds error text to the `errorInfo` variable.

- `Tcl_AddErrorInfo(Tcl_Interp *`*`interp`*`, char *`*`message`*`)`

Like `Tcl_AddObjErrorInfo`, but initializes `errorInfo` from the string value of the interpreter’s result if the error result is new.

- `Tcl_SetObjErrorCode(Tcl_Interp *`*`interp`*`,`
  `Tcl_Obj *`*`errorObjPtr`*`)`

Sets the `errorCode` variable from *`errorObjPtr`*.

- `Tcl_SetErrorCode(Tcl_Interp *`*`interp`*`, char *`*`element`*`,`
  `char *`*`element`*`, ... (char *) NULL)`

Sets the `errorCode` variable from a series of strings.

- `Tcl_SetErrorCodeVA(Tcl_Interp *`*`interp`*`, va_list `*`argList`*`)`

Sets the `errorCode` variable from a `va_list`.

- `Tcl_Obj *Tcl_GetReturnOptions(Tcl_Interp *`*`interp`*`,`
  `int `*`code`*`)`

Retrieves the dictionary of return options from an interpreter following a script evaluation.

- `int Tcl_SetReturnOptions(Tcl_Interp *`*`interp`*`,`
  `Tcl_Obj *`*`options`*`)`

Sets the return options of *`interp`* to be *`options`*.

- `CONST char *Tcl_PosixError(Tcl_Interp *`*`interp`*`)`

Sets the `errorCode` variable after an error in a POSIX kernel call. It reads the value of the `errno` C variable and calls `Tcl_SetErrorCode` to set `errorCode` in the POSIX format. Returns a human-readable diagnostic message for the error.

- `void Tcl_LogCommandInfo(Tcl_Interp *interp,`
  `CONST char *script, CONST char *command,`
  `int commandLength)`

Adds information about the command that was being executed when the error occurred to the `errorInfo` variable, and the line number stored internally in the interpreter is set.

- `void Tcl_Panic(CONST char *format, arg, arg, ...)`

Prints the formatted error message to the standard error file of the process, and then calls `abort` to terminate the process. `Tcl_Panic` does not return.

- `void Tcl_PanicVA(CONST char *format, va_list argList)`

Like `Tcl_Panic`, but takes a `va_list` argument.

- `void Tcl_SetPanicProc(Tcl_PanicProc panicProc)`

Sets the function to use when `Tcl_Panic` is called.

38.2 Completion Codes

Table 38.1 lists the full set of completion codes (which are integer return values) defined by Tcl. If a command procedure returns anything other than `TCL_OK`, Tcl aborts the evaluation of the script containing the command and returns the code as the result of `Tcl_EvalObjEx` (or `Tcl_EvalFile`, etc.). `TCL_OK` and `TCL_ERROR` have already been discussed; they are used for normal returns and errors respectively. The completion code `TCL_BREAK` or `TCL_CONTINUE` occurs if a `break` or `continue` command is invoked by a script; in both of these cases the interpreter's result is an empty string. The `TCL_RETURN` completion code occurs if `return` is invoked; in this case the interpreter's result is the intended result of the enclosing procedure. Additional completion codes may be defined by application-specific commands.

Table 38.1 Completion Codes Returned by Functions That Evaluate Scripts

Completion code	Meaning
TCL_OK	Command completed normally.
TCL_ERROR	Unrecoverable error occurred.
TCL_BREAK	break command was invoked.
TCL_CONTINUE	continue command was invoked.
TCL_RETURN	return command was invoked.
anything else	Defined by application.

As an example of how to generate a TCL_BREAK completion code, here is the command procedure for the break command:

```
int Tcl_BreakObjCmd(dummy, interp, objc, objv)
    ClientData dummy;       /* Not used. */
    Tcl_Interp *interp;     /* Current interpreter. */
    int objc;               /* Number of arguments. */
    Tcl_Obj *CONST objv[];  /* Argument objects. */
{
    if (objc != 1) {
        Tcl_WrongNumArgs(interp, 1, objv, NULL);
        return TCL_ERROR;
    }
    return TCL_BREAK;
}
```

In reality, in code that has been bytecode-compiled, things work a little differently, and the command gets called only if compilation doesn't take place. However, it is a good example.

TCL_BREAK, TCL_CONTINUE, and TCL_RETURN are used to unwind nested script evaluations back to an enclosing looping command or procedure invocation. Under most circumstances a procedure that receives any completion code other than TCL_OK from Tcl_EvalObjEx should immediately return that same completion code to its caller without modifying the interpreter's result. However, a few commands process some of the special completion codes without returning them upward. For example, here is an implementation of the while command:

```
int Tcl_WhileObjCmd(dummy, interp, objc, objv)
    ClientData dummy;        /* Not used. */
    Tcl_Interp *interp;      /* Current interpreter. */
    int objc;                /* Number of arguments. */
    Tcl_Obj *CONST objv[];  /* Argument objects. */
{
    int result, value;
    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv,
           "test command");
        return TCL_ERROR;
    }
    while (1) {
        /* Get the results of the expression which
           indicates whether to continue or not. */
        result = Tcl_ExprBooleanObj(interp, objv[1],
                                 &value);
        if (result != TCL_OK)
            return result;
        }
        if (value == 0) {
            return TCL_OK;
        }
        /* Evaluate the while body. */
        result = Tcl_EvalObjEx(interp, objv[2], 0);
        if (result == TCL_CONTINUE) {
            continue;
        } else if (code == TCL_BREAK) {
            return TCL_OK;
        } else if (code != TCL_OK) {
            return code;
        }
    }
}
```

After checking its argument count, `Tcl_WhileObjCmd` enters a loop where each iteration evaluates the command's first argument as an expression and its second argument as a script. If an error occurs while the expression is being evaluated, `Tcl_WhileObjCmd` returns the error. If the expression evaluates successfully but its value is `0`, the command terminates with a normal return. Otherwise, it evaluates the script argument. If the completion code is `TCL_CONTINUE`, `Tcl_WhileObjCmd` goes on to the next loop iteration. If the code is `TCL_BREAK`, `Tcl_WhileObjCmd` ends the execution of the command and returns `TCL_OK` to its caller. If `Tcl_EvalObjEx` returns any other completion code besides `TCL_OK`, `Tcl_WhileObjCmd` simply reflects that code upward. This causes the proper unwinding to occur on the `TCL_ERROR` or `TCL_RETURN` code and it also unwinds if any new completion codes are added in the future.

If an exceptional return unwinds all the way through the topmost script being evaluated, Tcl checks the completion code to be sure it is either TCL_OK or TCL_ERROR. If another code is returned, Tcl turns the return into an error with an appropriate error message. Furthermore, if a TCL_BREAK or TCL_CONTINUE exception unwinds all the way out of a procedure, Tcl also turns it into an error; for example:

```
  break
⇒ invoked "break" outside of a loop
  proc badbreak {} {break}
  badbreak
⇒ invoked "break" outside of a loop
```

Thus, applications need not worry about completion codes other than TCL_OK and TCL_ERROR when evaluating scripts from the topmost level. It is possible to circumvent this behavior with the Tcl_AllowExceptions function, covered in the reference documentation.

38.3 Setting errorCode

The last piece of information set after an error is the errorCode variable, which provides information about the error in a form that's easy to process with Tcl scripts—errorInfo (see below) is really meant to be read by people, not programs. It is intended for use in situations where a script is likely to catch the error, determine what went wrong, and attempt to recover from it if possible. If a command procedure returns an error to Tcl without setting errorCode, Tcl sets it to NONE. If a command procedure wishes to provide information in errorCode, it should invoke either Tcl_SetErrorCode or Tcl_SetObjErrorCode before returning TCL_ERROR.
Tcl_SetErrorCode takes as arguments an interpreter, any number of string arguments, and then a null pointer. It forms the strings into a list and stores the list as the value of errorCode. For example, suppose that you have written several commands to implement gizmo objects, and that there are several errors that could occur in commands that manipulate the objects, such as an attempt to use a nonexistent object. If one of your command procedures detects a nonexistent object error, it might set errorCode as follows:

```
Tcl_SetErrorCode(interp, "GIZMO", "EXIST",
                 "no object by that name",
                 (char *) NULL);
```

This will leave the value `GIZMO EXIST {no object by that name}` in `errorCode`. `GIZMO` identifies a general class of errors (those associated with gizmo objects), `EXIST` is the symbolic name for the particular error that occurred, and the last element of the list is a human-readable error message. You can store whatever you want in `errorCode` as long as the first list element doesn't conflict with other values already in use, but the overall idea is to provide symbolic information that can easily be processed by a Tcl script. For example, a script that accesses gizmos might catch errors, and if the error is a nonexistent gizmo, it might automatically create a new gizmo.
The object version is `Tcl_SetObjErrorCode`:

```
Tcl_SetObjErrorCode(Tcl_Interp *interp, Tcl_Obj
                                   *errorListObj);
```

where `errorListObj` is a Tcl list similar in form to the list that `Tcl_SetErrorCode` takes:

```
Tcl_Obj *err;
err = Tcl_NewObj();
Tcl_ListObjAppendElement(
    interp, err, Tcl_NewStringObj("GIZMO", -1));
Tcl_ListObjAppendElement(
    interp, err, Tcl_NewStringObj("EXISTS", -1));
Tcl_ListObjAppendElement(
    interp, err, Tcl_NewStringObj(
                     "no object by that name", -1));
Tcl_SetObjErrorCode(interp, err);
```

It's probably easier to use the non-object version in this case unless you need to directly insert numbers into the list. One would hope that errors are rare enough that the performance penalty incurred by the non-object code is not noticeable.

38.4 Managing the Return Options Dictionary

As mentioned in Chapter 13, it's possible to provide a return options dictionary to the `return` command in order to pass arbitrary information when raising an exception. The `catch` command can then retrieve the return options dictionary and process it in any way desired.
The equivalent method of accomplishing the same thing from within C code is to use `Tcl_SetReturnOptions` to set the return options dictionary and `Tcl_GetReturnOptions` to retrieve it.

A typical usage for Tcl_GetReturnOptions is to retrieve the stack trace when script evaluation returns an error, like so:

```
int code = Tcl_Eval(interp, script);
if (code == TCL_ERROR) {
    Tcl_Obj *options = Tcl_GetReturnOptions(interp, code);
    Tcl_Obj *key = Tcl_NewStringObj("-errorinfo", -1);
    Tcl_Obj *stackTrace;
    Tcl_IncrRefCount(key);
    Tcl_DictObjGet(NULL, options, key, &stackTrace);
    Tcl_DecrRefCount(key);
    /* Do something with stackTrace. */
}
```

This takes the dictionary object options and returns the value of the -errorinfo key, which should be a stack trace detailing what went wrong when script was evaluated.

38.5 Adding to the Stack Trace in errorInfo

When an error occurs, Tcl adds a description of it to the stack trace of the commands that were being evaluated at the time of the error; this stack trace is visible to the user in the errorInfo global variable. Tcl accomplishes this by calling the procedure Tcl_AddObjErrorInfo or Tcl_AddErrorInfo, both of which have the following prototypes:

```
void Tcl_AddObjErrorInfo(Tcl_Interp *interp,
                    char *message, int length);
void Tcl_AddErrorInfo(Tcl_Interp *interp,
                    char *message);
```

The difference is that *message* in Tcl_AddObjErrorInfo can contain embedded nulls. For most purposes, it’s simpler to just use Tcl_AddErrorInfo.
The first call to Tcl_AddErrorInfo after an error has occurred sets errorInfo to the error message stored in the interpreter and then appends message. Each subsequent call for the same error appends message to errorInfo’s current value. Whenever a command procedure returns TCL_ERROR, Tcl_Eval calls Tcl_AddErrorInfo to log information about the command that was being executed. If there are nested calls to Tcl_Eval, each one adds information about its command as it unwinds, so that a stack trace forms in errorInfo.
Command procedures can call Tcl_AddErrorInfo themselves to provide additional information about the context of the error. This is particularly

useful for command procedures that invoke `Tcl_Eval` recursively. For example, consider the following Tcl procedure, which is a buggy attempt to find the length of the longest element in a list:

```
proc longest {list} {
    set i [llength $list]
    while {$i >= 0} {
        set length [string length [lindex $list $i]]
        if {$length > $max} {
            set max $length
        }
        incr i -1
    }
    return $max
}
```

This procedure fails because it never initializes the variable `max`, so an error occurs when the `if` command attempts to read it. If the procedure is invoked with the command `longest {a 12345 xyz}`, the following stack trace is stored in `errorInfo` after the error:

```
can't read "max": no such variable
    while executing
"if {$length > $max} {
    set max $length
        }"
    (procedure "longest" line 5)
    invoked from within
"longest {a 12345 xyz}"
```

All of the information is provided by `Tcl_Eval` except for the line with comments in parentheses. The parenthesized message was generated by the code that evaluates procedure bodies.
Keen observers will note that there is no error message for the `while` command in this case. This is because it has been byte-compiled and thus doesn’t really exist as a normal procedure. If we could force it to be called as a procedure, we would get the following error trace:

```
can't read "max": no such variable
    while executing
"if {$length > $max} {
        set max $length
        }"
    ("while" body line 3)
    invoked from within
"$whilecmd {$i >= 0} {
        set length [string length [lindex $list $i]]
        if {$length > $max} {
            set max $length
        }
        incr i -1
       }"
    (procedure "longest_with_while" line 4)
    invoked from within
"longest_with_while {a 12345 xyz}"
```

Without the bytecode-compiled `while` command, we get a second parenthesized message, which gives us information on where the error occurred within the `while` body.

If you used the implementation of `while` from Section 38.2, instead of the built-in Tcl implementation, the first parenthesized message would be missing. The following C code is a replacement for the `while` loop and the code that follows it in `Tcl_WhileObjCmd`; it uses `Tcl_AppendResult` to add the parenthetical remark:

```
...
while (1) {
  /* Get the results of the expression which indicates
     whether to continue or not. */
  result = Tcl_ExprBooleanObj(interp, objv[1], &value);
  if (result != TCL_OK) {
    return result;
  }
  if (!value) {
    break;
  }

  /* Evaluate the while body. */
  result = Tcl_EvalObjEx(interp, objv[2], 0);
  if ((result != TCL_OK) &&
      (result != TCL_CONTINUE)) {
    if (result == TCL_ERROR) {

      Tcl_Obj *options = Tcl_GetReturnOptions(interp, code);
      Tcl_Obj *key = Tcl_NewStringObj("-errorline", -1);
      Tcl_Obj *value;
      int line;



      Tcl_DictObjGet(NULL, options, key, &value);
      Tcl_GetIntFromObj(NULL, value, &line);
      Tcl_DecrRefCount(key);
      Tcl_DecrRefCount(options);

      char msg[32 + TCL_INTEGER_SPACE];
      sprintf(msg, "\n    (\"while\" body line %d)", line);
      Tcl_AddErrorInfo(interp, msg);
    }

    break;
  }
}

if (result == TCL_BREAK) {
  result = TCL_OK;
}
if (result == TCL_OK) {
  Tcl_ResetResult(interp);
}
return result;
```

The `-errorline` key of the return options dictionary is set by `Tcl_EvalObjEx` whenever a command procedure returns `TCL_ERROR`; it gives the line number of the command that produced the error. A line number of 1 corresponds to the first line of the script being evaluated, which is the line containing the open brace in this example; the `if` command that generated the error is on line 3.

Note

Prior to Tcl 8.5, the error line number was available only by accessing `interp->errorLine` directly. This technique is strongly deprecated in Tcl 8.5 and may be disabled altogether in Tcl 8.6. Any existing code that accesses `interp->errorLine` should be replaced with code like that shown previously to access the `-errorline` key from the return options dictionary. Future versions of Tcl might provide more direct accessors for the line number information; consult the Tcl reference documentation for more information.

For simple Tcl commands you shouldn't need to invoke `Tcl_AddErrorInfo`: the information provided by `Tcl_Eval` is sufficient. However, if you write code that calls `Tcl_Eval`, it's a good idea to call `Tcl_AddErrorInfo` whenever `Tcl_Eval` returns an error, to provide information about why `Tcl_Eval` was invoked and also to include the line number of the error.
You must call `Tcl_AddErrorInfo` rather than trying to set the `errorInfo` variable directly, because `Tcl_AddErrorInfo` contains special code to detect the first call after an error and clear out the old contents of `errorInfo`.

38.6 `Tcl_Panic`

When the Tcl system encounters an error it can't handle—running out of memory, for example—it calls the `Tcl_Panic` function. Normally, this prints an error message to the standard error channel and calls `abort` to end the current process. You can use it in your own extensions like so:

Tcl_Panic(CONST char* *format*, *arg*, *arg*, *arg*, ...);

`format` is a format string like that in `printf`, and the *`arg`*s are arguments to print out. This function does not return, so if you need to call it from your extension, be sure it's the last thing you call. Do not use this function where an ordinary error would suffice, but only in cases where the application should halt immediately.

When Tcl is embedded in an application, you might wish to provide a more graceful way of shutting down Tcl and leave the process running, or simply let the application shut down according to its own rules. Of course, once `Tcl_Panic` has been called, you must ensure that your program does not attempt to make further use of Tcl. It is possible to register a function to be called in place of the default `PanicProc` using `Tcl_SetPanicProc`:

```
static void
My_Panic TCL_VARARGS_DEF(CONST char *, arg1) {
    ...
    Application-specific shutdown
    ...
    fprintf(stderr,
       "Tcl has panicked, terminating application\n");
    abort();
}

...

Tcl_SetPanicProc(My_Panic);
```

This registers a function that takes care of any cleanup specific to the application, prints an error message, and then terminates the process.

39. String Utilities

Tcl provides a number of utility functions that make working with strings and hash tables considerably easier and that C programmers can mostly use (with the exception of the encoding framework and the regular expression engine) independently of the rest of the Tcl library. This chapter describes Tcl's library functions for manipulating strings, including a dynamic string mechanism that allows you to build up arbitrarily long strings; a function for doing simple string matching; a suite of functions for working with Unicode characters and UTF-8 strings; and a function for testing the syntactic completeness of a command.

39.1 Functions Presented in This Chapter

The following string utility functions are described in this chapter:

- `void Tcl_DStringInit(Tcl_DString *dsPtr)`

Initializes `dsPtr`'s value to an empty string (the previous contents of `dsPtr` are discarded without cleaning up).

- `char *Tcl_DStringAppend(Tcl_DString *dsPtr,`
  `const char *string, int length)`

Appends `length` bytes from `string` to `dsPtr`'s value and returns the new value of `dsPtr`. If `length` is less than 0, appends all of `string`.

- `char *Tcl_DStringAppendElement(Tcl_DString *dsPtr,`
  `const char *string)`

Converts `string` to a proper list element and appends it to `dsPtr`'s value (with separator space if needed). Returns the new value of `dsPtr`.

- `Tcl_DStringStartSublist(Tcl_DString *dsPtr)`

Adds suitable bytes to `dsPtr` ({, for example) to initiate a sublist.

- `Tcl_DStringEndSublist(Tcl_DString *dsPtr)`

Adds suitable bytes to `dsPtr` (}, for example) to terminate a sublist.

- `char *Tcl_DStringValue(Tcl_DString *dsPtr)`

Returns a pointer to the current value of `dsPtr`.

- `int Tcl_DStringLength(Tcl_DString *dsPtr)`

Returns the number of bytes in `dsPtr`'s value, not including the terminating null byte.

- `Tcl_DStringTrunc(Tcl_DString *dsPtr, int newLength)`

If `dsPtr` has more than `newLength` bytes, shortens it to include only the first `newLength` bytes.

- `Tcl_DStringFree(Tcl_DString *dsPtr)`

Frees up any memory allocated for `dsPtr` and re-initializes `dsPtr`'s value to an empty string.

- `Tcl_DStringResult(Tcl_Interp *interp, Tcl_DString *dsPtr)`

Moves the value of `dsPtr` to the interpreter's result and re-initializes `dsPtr`'s value to an empty string.

- `Tcl_DStringGetResult(Tcl_Interp *interp,`
```
                     Tcl_DString *dsPtr)
```

Moves the contents of the interpreter's result into `dsPtr` and re-initializes the interpreter's result to the empty string.

- `int Tcl_StringMatch(const char *string,`
```
                     const char *pattern)
```

Returns `1` if `string` matches `pattern` using glob-style rules for pattern matching, `0` otherwise.

- `int Tcl_StringCaseMatch(const char *string,`
```
                     const char *pattern, int nocase)
```

Returns `1` if `string` matches `pattern` using glob-style rules for pattern matching, `0` otherwise. If `nocase` is `0`, the match is performed case-sensitively, and if it is `1`, the match is performed case-insensitively.

- `int Tcl_RegExpMatchObj(Tcl_Interp *interp,`
```
                     Tcl_Obj *textObj, Tcl_Obj *patObj)
```

Tests whether the regular expression in `patObj` matches the string in `textObj`, returning `1` if they match, `0` otherwise. If an error occurs while `patObj` is compiling, an error message is left in the interpreter (if non-null) and `-1` is returned.

- `int Tcl_RegExpMatch(Tcl_Interp *interp, char *text,`
```
                     const char *pattern)
```

Tests whether the regular expression in `pattern` matches the string in `text`, returning `1` if they match, `0` otherwise. If an error occurs while `pattern` is compiling, an error message is left in the interpreter (if non-null) and `-1` is returned.

- `Tcl_RegExp Tcl_RegExpCompile(Tcl_Interp *interp,`
```
                         const char *pattern)
```

Compiles `pattern` to a regular expression and returns a handle to it. If an error occurs while `pattern` is compiling, an error message is left in the interpreter (if non-null) and `NULL` is returned.

- `int Tcl_RegExpExec(Tcl_Interp *interp, Tcl_RegExp regexp,`
```
                     char *text, char *start)
```

Tests whether the compiled regular expression *regexp* matches the string *text*, which itself should be a substring of the string *start* (information necessary for repeated matching). Returns `1` if there is a match, `0` if there is no match, and `-1` if an error occurred (with an error message in the interpreter if that is non-null).

- `Tcl_RegExpRange(Tcl_RegExp regexp, int index,`
  `const char **startPtr, const char **endPtr)`

Provides additional information about a subexpression range *index* after a successful match against the expression *regexp*. The *startPtr* and *endPtr* arguments indicate variables that are set to point to the start and end of the matched range respectively.

- `Tcl_RegExp Tcl_GetRegExpFromObj(Tcl_Interp *interp,`
  `Tcl_Obj *patObj, int cflags)`

Compiles the pattern in *patObj* to a regular expression, given the compilation flags in *cflags*. If an error occurs during compiling, it returns `NULL` and stores an error message in the interpreter. See the reference documentation for a description of the supported compilation flags.

- `int Tcl_RegExpExecObj(Tcl_Interp *interp,`
  `Tcl_RegExp regexp, Tcl_Obj *textObj, int offset,`
  `int nmatches, int eflags)`

Matches a regular expression *regexp* against a string in *textObj*, starting from *offset* characters into the string. The *nmatches* argument describes the maximum number of subexpressions for which information will be requested, and *eflags* gives a number of regular-expression execution flags. Returns `1` if the expression matches, `0` if it does not. If an error occurs during matching, it returns `-1` and leaves an error message in the interpreter. See the reference documentation for a description of the supported execution flags.

- `Tcl_RegExpGetInfo(Tcl_RegExp regexp,`
  `Tcl_RegExpInfo *infoPtr)`

Obtains extended information about the last match of the regular expression *regexp*. The information describes the number of possible subexpressions, the number that actually matched, the character location within the string where each subexpression matched, and (with correct compilation flags) whether it is possible to match the regular expression at all, even with additional text.

- `char *Tcl_ExternalToUtfDString(Tcl_Encoding encoding,`
  `const char *src, int srcLen, Tcl_DString *dsPtr)`

Converts the character data in *src* (of length *srcLen*, or up to the first zero byte if *srcLen* is negative) from the *encoding* encoding to the UTF-8 encoding used internally in Tcl and returns the resulting string. The buffer in *dsPtr* is used as a workspace; it should be released with `Tcl_DStringFree` once the converted string is no longer required. If *encoding* is `NULL`, the current system encoding is used.

- `char *Tcl_UtfToExternalDString(Tcl_Encoding encoding, const char *src, int srcLen, Tcl_DString *dsPtr)`

Converts the character data in *src* (of length *srcLen*, or up to the first zero byte if *srcLen* is negative) from the UTF-8 encoding used internally in Tcl to the *encoding* encoding and returns the resulting string. The buffer in *dsPtr* is used as a workspace; it should be released with `Tcl_DStringFree` once the converted string is no longer required. If *encoding* is `NULL`, the current system encoding is used.

- `Tcl_Encoding Tcl_GetEncoding(Tcl_Interp *interp, const char *name)`

Returns the encoding called *name*, or if no encoding with that name is known, returns `NULL` and places an error message in the interpreter's result. The returned encoding should be released with `Tcl_FreeEncoding` when it is no longer required.

- `Tcl_FreeEncoding(Tcl_Encoding encoding)`

Releases an encoding previously returned by `Tcl_GetEncoding`.

- `int Tcl_UniCharToUtf(int ch, char *buf)`

Converts *ch* to UTF-8 and stores it in the buffer at *buf*. Returns the number of bytes written.

- `int Tcl_UtfToUniChar(const char *src, Tcl_UniChar *chPtr)`

Converts the first UTF-8 character at *src* to Unicode and writes it to the variable pointed to by *chPtr*. Returns the number of bytes read.

- `char *Tcl_UniCharToUtfDString(const Tcl_UniChar *uniStr, int uniLength, Tcl_DString *dsPtr)`

Converts the string of *uniLength* Unicode characters at *uniStr* to UTF-8 and stores that in the given `Tcl_DString`. Returns the UTF-8 string produced.

- `Tcl_UniChar *Tcl_UtfToUniCharDString(const char *src, int length, Tcl_DString *dsPtr)`

Converts the string of up to *length* UTF-8 characters at *src* to a Unicode string that is stored in the given `Tcl_DString`. Returns the Unicode string produced. If *length* is `-1`, all UTF-8 characters up to the end of the input string are processed.

- `int Tcl_UniCharLen(const Tcl_UniChar *uniStr)`

Returns the length (in Unicode characters) of the null-terminated Unicode string in *uniStr*.

- `int Tcl_UniCharNcmp(const Tcl_UniChar *ucs, const Tcl_UniChar *uct, int numChars)`

Compares the first *numChars* characters of two Unicode strings for equality, returning `0` if they are the same, a negative number if *`ucs`* has the smallest character at the first place they differ, or a positive number if *`uct`* has the smallest character at the first place they differ.

- `int Tcl_UniCharNcasecmp(const Tcl_UniChar *`*`ucs`*`,`
  `const Tcl_UniChar *`*`uct`*`, int `*`numChars`*`)`

Case-insensitively compares the first *`numChars`* characters of two Unicode strings for equality, returning `0` if they are the same, a negative number if *`ucs`* has the smallest character at the first place they differ, or a positive number if *`uct`* has the smallest character at the first place they differ.

- `int Tcl_UniCharCaseMatch(const Tcl_UniChar *`*`uniStr`*`,`
  `const Tcl_UniChar *`*`uniPattern`*`, int `*`nocase`*`)`

Returns `1` if *`uniStr`* matches *`uniPattern`* using glob-style rules for pattern matching, `0` otherwise. If *`nocase`* is non-zero, the pattern is compared case-insensitively; otherwise it is compared case-sensitively.

- `int Tcl_UtfNcmp(const char *`*`cs`*`, const char *`*`ct`*`,`
  `int `*`numChars`*`)`

Compares the first *`numChars`* characters of two UTF-8-encoded strings for equality, returning `0` if they are the same, a negative number if *`cs`* has the smallest character at the first place they differ, or a positive number if *`ct`* has the smallest character at the first place they differ.

- `int Tcl_UtfNcasecmp(const char *`*`cs`*`, const char *`*`ct`*`,`
  `int `*`numChars`*`)`

Case-insensitively compares the first *`numChars`* characters of two UTF-8-encoded strings for equality, returning `0` if they are the same, a negative number if *`cs`* has the smallest character at the first place they differ, or a positive number if *`ct`* has the smallest character at the first place they differ.

- `int Tcl_UtfCharComplete(const char *`*`src`*`, int `*`length`*`)`

Tests if the *`length`* bytes (or all the bytes up to the next zero byte if length is `-1`) at *`src`* represent at least one UTF-8 character.

- `int Tcl_NumUtfChars(const char *`*`src`*`, int `*`length`*`)`

Returns the number of UTF-8 characters in the string at *`src`*. Only the first *`length`* bytes are used, or all bytes up to the first null are used if *`length`* is `-1`.

- `const char *Tcl_UtfFindFirst(const char *`*`src`*`, int `*`ch`*`)`

Returns the first instance of the character *`ch`* in the null-terminated UTF-8-encoded string *`src`*, or `NULL` if the character is not present.

- `const char *Tcl_UtfFindLast(const char *`*`src`*`, int `*`ch`*`)`

Returns the last instance of the character *`ch`* in the null-terminated UTF-8-encoded string *`src`*, or `NULL` if the character is not present.

- `const char *Tcl_UtfNext(const char *`*`src`*`)`

Returns the location of the next UTF-8 character in the string *`src`*.

- `const char *Tcl_UtfPrev(const char *`*`src`*`,`
  `const char *`*`start`*`)`

Returns the location of the previous UTF-8 character before `src`, where the start of the string is pointed to by `start`.

- `Tcl_UniChar Tcl_UniCharAtIndex(const char *`*`src`*`, int `*`index`*`)`

Returns the `index`th Unicode character extracted from the UTF-8 string `src`. The string must contain at least `index` UTF-8 characters. The index must not be negative.

- `const char *Tcl_UtfAtIndex(const char *`*`src`*`, int `*`index`*`)`

Returns a pointer to the `index`th UTF-8 character in the UTF-8 string `src`. The string must contain at least `index` UTF-8 characters.

- `int Tcl_UtfBackslash(const char *`*`src`*`, int *`*`readPtr`*`,`
  `char *`*`dst`*`)`

Parses a Tcl backslash sequence in `src`, storing the ensuing UTF-8 character in the string at `dst`. The variable pointed to by `readPtr` is updated to contain how many bytes of `src` were parsed. The number of bytes written to `dst` is returned.

- `int Tcl_CommandComplete(const char *`*`cmd`*`)`

Returns `1` if `cmd` holds one or more syntactically complete commands, `0` if the last command in `cmd` is incomplete because of open braces or other faults.

39.2 Dynamic Strings

A *dynamic string* is a string that can be appended to without bound. As you append information to a dynamic string, Tcl automatically enlarges the memory area allocated for it as needed. If the string is short, Tcl avoids memory allocation altogether by using a small static buffer to hold the string. It also forms a convenient system for managing buffers dynamically, since it takes care of the low-level memory management details for you. Tcl provides 11 functions and macros for manipulating dynamic strings:

- `Tcl_DStringInit` initializes a dynamic string to an empty string. Note that a dynamic string that has been initialized and not appended to does not need to be passed to `Tcl_DStringFree`.
- `Tcl_DStringAppend` adds bytes to a dynamic string. The dynamic string remains null-terminated.
- `Tcl_DStringAppendElement` adds a new list element to a dynamic string.
- `Tcl_DStringStartSublist` and `Tcl_DStringEndSublist` are used to create sublists within a dynamic string.
- `Tcl_DStringValue` returns the current value of a dynamic string.

- `Tcl_DStringLength` returns the current length of a dynamic string.
- `Tcl_DStringTrunc` truncates a dynamic string.
- `Tcl_DStringFree` releases any storage allocated for a dynamic string and re-initializes the string.
- `Tcl_DStringResult` moves the value of a dynamic string to the result for an interpreter and re-initializes the dynamic string. `Tcl_DStringGetResult` does the reverse operation, transferring the result to the dynamic string and resetting the interpreter.

The following code uses several of these functions to implement a `map` command, which takes a list and generates a new list by applying some operation to each element of the original list. The `map` command takes two arguments: a list and a Tcl command. For each element in the list, it executes the given command with the list element appended as an additional argument. With the results of all the commands it generates a new list, and then returns this list as its result. Here are some examples of how you might use the `map` command:

```
  proc inc {x} {expr {$x + 1}}
  map {4 18 16 19 -7} inc
⇒ 5 19 17 20 -6
  proc addz {x} {concat $x z}
  map {a b {a b c}} addz
⇒ {a z} {b z} {a b c z}
```

Here is a command function that implements `map`:

```
int MapCmd(ClientData clientData, Tcl_Interp *interp,
      int argc, char *argv[]) {
   Tcl_DString command, newList;
   int listArgc, i, result;
   char **listArgv;

   if (argc != 3) {
      Tcl_SetResult(interp, "wrong # args",
            TCL_STATIC);
      return TCL_ERROR;
   }
   if (Tcl_SplitList(interp, argv[1], &listArgc,
         &listArgv) != TCL_OK) {
      return TCL_ERROR;
   }
   Tcl_DStringInit(&newList);
   Tcl_DStringInit(&command);
   for (i=0 ; i<listArgc ; i++) {
      Tcl_DStringAppend(&command, argv[2], -1);
      Tcl_DStringAppendElement(&command, listArgv[i]);
      result = Tcl_Eval(interp,
            Tcl_DStringValue(&command));
      Tcl_DStringFree(&command);
      if (result != TCL_OK) {
         Tcl_DStringFree(&newList);
         ckfree((char *) listArgv);
         return result;
      }
      Tcl_DStringAppendElement(&newList,
            Tcl_GetResult(interp));
   }
   Tcl_DStringResult(interp, &newList);
   ckfree((char *) listArgv);
   return TCL_OK;
}
```

`MapCmd` uses two dynamic strings. One holds the result list and the other holds the command to execute in each step. The first dynamic string is needed because the length of the command is unpredictable, and the second one is needed to store the result list as it builds up (this information cannot be kept immediately in the interpreter's result because the result will be overwritten by the command that is evaluated to process the next list element). Each dynamic string is represented by a structure of type `Tcl_DString`. The structure holds information about the string, such as a pointer to its current value, a small array to use for small strings, and a length. Tcl does not allocate `Tcl_DString` structures; it is up to you to allocate the structure (as a local variable, for example) and pass its address to the dynamic string library functions. You should never access the fields of a `Tcl_DString` structure directly; use the macros and functions provided by Tcl.

After checking its argument count, extracting all of the elements from the initial list, and initializing its dynamic strings, `MapCmd` enters a loop to process the elements of the list. For each element, it first creates the command to execute for that element. It does this by calling `Tcl_DStringAppend` to append the part of the command provided in `argv[2]`, then it calls `Tcl_DStringAppendElement` to append the list element as an additional argument. These functions are similar in that both add new information to the dynamic string. However, `Tcl_DStringAppend` adds the information as raw text, whereas `Tcl_DStringAppendElement` converts its string argument to a proper list element and adds that list element to the dynamic string (with a separator space, if needed).

In this case, it is important to use `Tcl_DStringAppendElement` for the list element so that it becomes a single word of the Tcl command under construction. If `Tcl_DStringAppend` were used instead and the element were `a b c` as in the example earlier in this section, the command passed to `Tcl_Eval` would be `addz a b c`, which would result in an error (because too many arguments are passed to the `addz` procedure). When `Tcl_DStringAppendElement` is used, the command is `addz {a b c}`, which parses correctly.[1]

[1]. There are other ways to invoke commands with extra arguments in a systematically correct way. For example, `Tcl_NewListObj` and the other list creation functions build a correctly quoted string that you can use with `Tcl_EvalObjEx`, and `Tcl_EvalObjv` allows the execution of a command without any parsing at all.

Once `MapCmd` has created the command to execute for an element, it invokes `Tcl_Eval` to evaluate the command. The `Tcl_DStringFree` call frees any memory allocated for the command string and resets the dynamic string to an empty value for use in the next command. If the command returns an error, `MapCmd` returns that same error; otherwise, it uses `Tcl_DStringAppendElement` to add the result of the command to the result list as a new element.

`MapCmd` calls `Tcl_DStringResult` after all of the list elements have been processed. This transfers the value of the string to the interpreter's result in an efficient way (for example, it might transfer ownership of a dynamically allocated buffer to the interpreter instead of copying the buffer).

Before returning, `MapCmd` must free any memory allocated for the dynamic strings. It turns out that `Tcl_DStringFree` has already done this for the command, and `Tcl_DStringResult` has done this for `newList`.

It is possible to implement much of `MapCmd` more efficiently through the use of the `Tcl_Obj` API. However, while that makes many parts of the processing more efficient, the actual dispatch mechanism mentioned (using a `Tcl_DString`) is still one of the simplest ways to achieve it. Indeed, the command might be written like this:

```
int MapCmd(ClientData clientData, Tcl_Interp *interp,
      int objc, Tcl_Obj *const objv[]) {
   Tcl_DString command;
   int listObjc, i, result;
   Tcl_Obj *newList, **listObjv;

   if (objc != 3) {
      Tcl_WrongNumArgs(interp, 1, objv,
            "list command");
      return TCL_ERROR;
   }
   if (Tcl_ListObjGetElements(interp, objv[1],
         &listArgc, &listObjv) != TCL_OK) {
      return TCL_ERROR;
   }
   newList = Tcl_NewObj();
   Tcl_DStringInit(&command);
   for (i=0 ; i<listObjc ; i++) {
      Tcl_DStringAppend(&command,
            Tcl_GetString(objv[2]), -1);


      Tcl_DStringAppendElement(&command,
            Tcl_GetString(listObjv[i]));
      result = Tcl_Eval(interp,
            Tcl_DStringValue(&command));
      Tcl_DStringFree(&command);
      if (result != TCL_OK) {
         Tcl_DecrRefCount(newList);
         return result;
      }
      Tcl_ListObjAppendElement(NULL, newList,
            Tcl_GetObjResult(interp));
   }
   Tcl_SetObjResult(interp, newList);
   return TCL_OK;
}
```

As you can see, much of the code is extremely similar despite the differences in function names.

39.3 String Matching

The functions Tcl_StringMatch and Tcl_StringCaseMatch (an extended version that allows control over whether matches are case-sensitive or not) provide the same functionality as the string match Tcl command. Given a string and a pattern, they return 1 if the string matches the pattern using glob-style matching and 0 otherwise. For example, here is a command function that uses Tcl_StringMatch to implement a simplified version of lsearch that provides only glob-style matching. It returns the index of the first element in a list that matches a pattern, or -1 if no element matches:

```
int LsearchCmd(ClientData clientData,
        Tcl_Interp *interp, int objc,
        Tcl_Obj *const objv[]) {
    char *pattern;
    int i, elemc;
    Tcl_Obj **elemv;
    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv,
                "list pattern");
        return TCL_ERROR;
    }
    if (Tcl_ListObjGetElements(interp, objv[1],
            &elemc, &elemv) != TCL_OK) {
        return TCL_ERROR;
    }

    pattern = Tcl_GetString(objv[2]);
    for (i=0 ; i<elemc ; i++) {
       if (Tcl_StringMatch(Tcl_GetString(elemv[i]),
               pattern)) {
           Tcl_SetObjResult(interp, Tcl_NewIntObj(i));
           return TCL_OK;
       }
    }
    Tcl_SetObjResult(interp, Tcl_NewIntObj(-1));
    return TCL_OK;
}
```

39.4 Regular Expression Matching

When the simple matching capabilities of Tcl_StringMatch are insufficient, it is usually necessary to use regular expressions instead. Chapter 5 described the syntax and capabilities of regular expressions at the Tcl scripting level; this section describes Tcl’s C API for working with regular expressions.

The simplest functions to use for regular expression matching are `Tcl_RegExpMatch` and `Tcl_RegExpMatchObj`. These functions (which differ only in the types of their arguments) test whether a regular expression pattern matches a given string. They allow us to rewrite our previous simple `lsearch`-like example to use regular expressions like this:

```
int RLsearchCmd(ClientData clientData,
        Tcl_Interp *interp, int objc,
        Tcl_Obj *const objv[]) {
    Tcl_Obj *pattern;
    int i, elemc, result;
    Tcl_Obj **elemv;
    if (objc != 3) {
       Tcl_WrongNumArgs(interp, 1, objv, "list pattern");
       return TCL_ERROR;
    }
    if (Tcl_ListObjGetElements(interp, objv[1],
          &elemc, &elemv) != TCL_OK) {
       return TCL_ERROR;
    }
    pattern = objv[2];
    for (i=0 ; i<elemc ; i++) {
       result = Tcl_RegExpMatchObj(interp, elemv[i], pattern);
       if (result == -1) {
          return TCL_ERROR;
       }

       if (result == 1) {
          Tcl_SetObjResult(interp, Tcl_NewIntObj(i));
          return TCL_OK;
       }
    }
    Tcl_SetObjResult(interp, Tcl_NewIntObj(-1));
    return TCL_OK;
}
```

The main additional complexity here is that the regular expression compiler can fail if presented with an invalid regular expression. When that is allowed for, the structure of the code that uses this is virtually identical to the version based on simple pattern matching.

When additional complexity is required, such as when applying the regular expression efficiently to large numbers of strings or the same string multiple times to obtain all the locations within it that match, or accessing to the subexpressions that were matched, the more sophisticated variants of the regular expression API are used. The first level of sophistication allows the separation of the compilation and execution phases of regular expression matching through `Tcl_RegExpCompile` and `Tcl_RegExpExec`. These functions also

allow control over where in the string to start matching. They additionally enable the use of `Tcl_RegExpRange` to identify the substrings that match the subexpressions of the regular expression.
A deeper level of sophistication is available as well. `Tcl_GetRegExpFromObj` allows control over the compilation flags used to generate a regular expression matcher, controlling things such as the regular expression dialect used or exactly how newlines within the string are processed. `Tcl_RegExpExecObj` allows control over not just the features that `Tcl_RegExpExec` does, but also over how many subexpressions are expected and whether the ends of strings should act as anchors. `Tcl_RegExpGetInfo` provides similar information to `Tcl_RegExpRange`, but in more detail and in a way better suited for use with the `Tcl_Obj` API.

Note

Be aware that when matching regular expressions, the `Tcl_Obj`-based APIs are considerably more efficient. This is because of the better treatment of caching afforded by the `Tcl_Obj` system, and it is especially true when large numbers of expressions are used. However, when small numbers of patterns are used, the difference is not particularly great because the regular expression engine maintains an internal per-thread cache of compiled regular expressions. Also note that in virtually all cases, both literal string comparison and `Tcl_StringMatch` are more efficient when the thing being matched is something that they are capable of matching; regular expressions should be used only where their considerably greater power is required.

39.5 Working with Character Encodings

A *character encoding* is a mapping of characters and symbols used in written language into a binary format used by computers. Internally, Tcl represents strings in the Unicode encoding using the UTF-8 format. However, when you need to transfer a string to some other non-Tcl API, you may very well need to transform the string into a different encoding. For example, if you are passing a string to an operating system function, you need to send that string in the system encoding (often ISO 8859-1 in North

America, but quite different in other locales); without that, many characters (such as copyright marks, quotes, many currency symbols, etc.) get interpreted as multiple characters, often with a confounding accented letter apparently added. Chapter 5 described commands at the Tcl scripting level; this section describes Tcl's C API for working with character encodings.

Note

Internally, Tcl represents zero bytes as a multibyte denormalized sequence because this allows UTF-8 strings to be processed correctly with normal C string utility functions. Note that only internal strings are represented this way; strings read from or presented to the outside world never use this encoding scheme.

Tcl provides several functions for converting between Tcl's UTF-8 representation and the world of external encodings. `Tcl_UtfToExternalDString` takes a Tcl internal string and converts it to a string in some other encoding that is stored in a `Tcl_DString`, and `Tcl_ExternalToUtfDString` goes the other way. In order to use either of these functions, you need to get a handle for the encoding with `Tcl_GetEncoding`. Pass the special value `NULL` instead of an encoding handle to use the system encoding (which is often the right thing to do, especially if using any operating system API). After using a handle for an encoding gotten with `Tcl_GetEncoding`, always release that handle with `Tcl_FreeEncoding`.

For example, the following function `LogCmd` is an implementation of a Tcl command that writes log messages to a POSIX syslog. After checking that the correct number of arguments has been supplied, it converts the supplied argument from Tcl's internal encoding into the system encoding with `Tcl_UtfToExternalDString` and writes it to the log as an informational message.

```
int LogCmd(ClientData clientData, Tcl_Interp *interp,
        int objc, Tcl_Obj *const objv[]) {
    Tcl_DString buffer;
    char *messageStr;

    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "message");
        return TCL_ERROR;
    }
    Tcl_DStringInit(&buffer);
    messageStr = Tcl_UtfToExternalDString(NULL,
            Tcl_GetString(objv[1]), -1, &buffer);
    openlog("TclExample", 0, LOG_USER);
    syslog(LOG_INFO, "%s", messageStr);
    closelog();
    Tcl_DStringFree(&buffer);
    return TCL_OK;
}
```

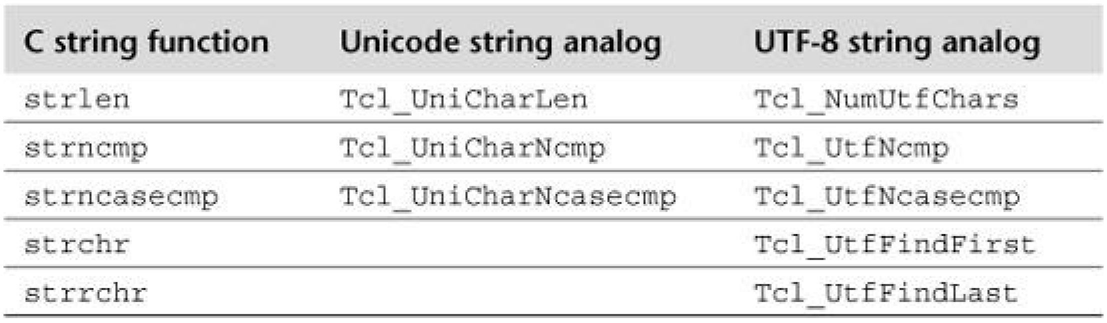
The other functions for working with encodings are `Tcl_UtfToExternal` and `Tcl_ExternalToUtf`. These functions are focused on providing efficient support for streaming transformations; when converting whole strings, the `Tcl_DString`-based API is much more convenient.
The most common case of having to work with encodings explicitly is when working with network sockets. This is because with the worldwide Internet, you cannot count on other parties to use the same encoding as you. Indeed, this is sufficiently important that all channels have their own encoding support, which means that as long as you configure the encoding upon them correctly, you do not explicitly need to manage the encoding. However, this can get distinctly complex with some protocols, especially those that mix different encoding schemes (for example, binary and a text encoding).

39.6 Handling Unicode and UTF-8 Strings

Though conceptually they use the same set of characters, Unicode strings (strictly, UCS-2 strings of host-system endian-ness) and UTF-8 strings have fundamentally different properties. In particular, Unicode strings can be indexed into at arbitrary offsets in constant time and so can have sophisticated operations applied to them rapidly, whereas UTF-8 strings are simple to pass through many traditional APIs. Tcl provides many functions for working with Unicode and UTF-8 strings.
The functions `Tcl_UniCharToUtf` and `Tcl_UtfToUniChar` provide basic mapping

between Unicode and UTF-8 on a character-by-character basis. These are extended to working with whole strings of Unicode and UTF-8 characters by the functions `Tcl_UniCharToUtfDString` and `Tcl_UtfToUniCharDString`, which both target a buffer in a `Tcl_DString`.
Tcl also provides analogs of many of the C string functions; Table 39.1 shows the mapping of C string functions to analogous Tcl functions. `Tcl_StringCaseMatch`, which handles UTF-8 strings, also has a Unicode string analog: `Tcl_UniCharCaseMatch`.

Table 39.1 Tcl Unicode and UTF-8 Analogs of C String Functions



C string function	Unicode string analog	UTF-8 string analog
strlen	Tcl_UniCharLen	Tcl_NumUtfChars
strncmp	Tcl_UniCharNcmp	Tcl_UtfNcmp
strncasecmp	Tcl_UniCharNcasecmp	Tcl_UtfNcasecmp
strchr		Tcl_UtfFindFirst
strrchr		Tcl_UtfFindLast

In addition, the functions `Tcl_UtfNext` and `Tcl_UtfPrev` are used to step forward and backward through a UTF-8-encoded string, and `Tcl_UniCharAtIndex` and `Tcl_UtfAtIndex` respectively allow looking up a character at a specific index of a Unicode or UTF-8 string, returning the Unicode character at that location or a pointer to that location respectively.
When processing a buffer that might contain partial UTF-8 characters, the function `Tcl_UtfCharComplete` returns whether the string its argument points to contains at least one complete UTF-8 character. This is useful because many protocols require that only complete characters be transferred in any message.
Finally, `Tcl_UtfBackslash` provides a parser for the backslash sequences supported by Tcl. It copies a backslash sequence from a source buffer to a target buffer, transforming it in the process to the character that it maps to according to Tcl's syntax rules and returning the number of bytes written into the target buffer in the process.
The C function that follows illustrates how to use these together. It takes a character and a pointer to a UTF-8 string and returns a new string that consists of that character and the two characters on either side of it. This could be implemented using the Tcl script

```
set i [string first $ch $string] expr
```

```
{$i<0 ? "" : [string range $string $i-2 $i+2]}
```

and is a simple application of `strchr` with ASCII strings, but it is a little more complex with UTF-8 because characters are not a constant width.

```
char *SubstringAbout(int ch, const char *string) {
    const char *end, *loc;
    char *buf;
    int i;

    /* Find the character. */
    loc = end = Tcl_UtfFindFirst(string, ch);
    if (loc == NULL) {
        return NULL;
    }
    /* Find one char past the end of the area. */
    for (i=0; *end!='\0' && i<3; i++) {
        end = Tcl_UtfNext(end);
    }
    /* Find the start of the area. */
    for (i=0; i<2; i++) {
        loc = Tcl_UtfPrev(loc, string);
    }
    /* Copy and return the substring. */
    buf = ckalloc((end-loc)+1);
    memcpy(buf, loc, end-loc);
    buf[end-loc] = '\0';
    return buf;
}
```

Note

Though the result string of the `SubstringAbout` function is only up to five characters long, those five characters could take as many as 15 bytes, depending on the writing system used. In the future, it could be even longer as the space of Unicode characters outside the Basic Multilingual Plane gets defined and supported. But since this function makes no assumptions about the actual length of the result string, it will continue to function correctly when this happens.

39.7 Command Completeness

When an application reads commands typed interactively, it must wait until a complete command has been entered before evaluating it. For example, suppose an application is reading commands from standard input and the user types the following three lines:

```
foreach i {1 2 3 4 5} {
    puts "$i*$i is [expr $i*$i]"
}
```

If the application reads each line separately and passes it to `Tcl_Eval`, the first line will generate a “missing close brace” error. Instead, the application should collect input until all the commands read are complete (for example, there are no unmatched braces or quotes), then execute all of the input as a single script. The function `Tcl_CommandComplete` makes this possible. It takes a string as argument and returns `1` if the string contains syntactically complete commands or `0` if the last command is not yet complete.
The C function that follows uses dynamic strings and `Tcl_CommandComplete` to read and evaluate a command typed on standard input. (For clarity, it avoids using the Tcl channel API. Otherwise, this code would be longer and would obscure the parts of the function that are really being demonstrated.) It collects input until all the commands read are complete, and then it evaluates the command(s) and returns the completion code from the evaluation. It uses `Tcl_RecordAndEval` to evaluate the command so that the command is recorded on the history list.

```
int DoOneCmd(Tcl_Interp *interp) {
    char line[200];
    Tcl_DString cmd;
    int code;
    Tcl_DStringInit(&cmd);
    while (1) {
        if (fgets(line, 200, stdin) == NULL) {
            break;
        }
        Tcl_DStringAppend(&cmd, line, -1);
        if (Tcl_CommandComplete(Tcl_DStringValue(&cmd))) {
            break;
        }
    }
    code = Tcl_RecordAndEval(interp,
            Tcl_DStringValue(&cmd), 0);
    Tcl_DStringFree(&cmd);
    return code;
}
```

In the `foreach` example, `DoOneCmd` will collect all three lines before evaluating them. If an end of file occurs, `fgets` will return `NULL`, and `DoOneCmd` will evaluate the command if it is not yet complete.

`Tcl_CommandComplete` checks for completeness only in the sense of parsing correctly. It does not guarantee that the script will behave correctly. For example, if a user accidentally splits a command like `set x y` over two lines by typing a newline after the `x`, each line will be considered to be complete. The first line will simply query the variable instead of modifying it, and the second line will invoke a command `y`, which will probably generate an error.

40. Hash Tables

A *hash table* is a collection of *entries*, where each entry consists of a *key* and a *value*. No two entries may have the same key. Given a key, a hash table can locate its entry very quickly and hence the associated value. Tcl contains a general-purpose hash table package that it uses in several places internally. For example, all of the commands in an interpreter are stored in a hash table where the key for each entry is a command name and the value is a pointer to information about the command. All of the variables in a namespace are stored in another hash table where the key for each entry is the name of a variable and the value is a pointer to information about the variable.

Tcl exports its hash table facilities through a set of library functions so that applications can use them, too. The most common use for hash tables is to associate names with objects. In order for an application to implement a new kind of object, it must give the objects textual names for use in Tcl commands. When a command function receives an object name as an argument, it must locate the C data structure for the object. Typically there is one hash table for each type of object, where the key for an entry is an object name and the value is a pointer to the C data structure that represents the object. When a command function needs to find an object, it looks up its name in the hash table. If there is no entry for the name, the command function returns an error.

The examples in this chapter use a hypothetical application that implements objects called "gizmos." Each gizmo is represented internally with a structure declared like this:

```
typedef struct Gizmo {
    ... fields of gizmo object ...
} Gizmo;
```

The application uses names like `gizmo42` to refer to gizmos in Tcl commands, where each gizmo has a different number at the end of its name. The application follows the action-oriented approach described in Section 30.3 by providing a collection of Tcl commands to manipulate the objects, such as `gizmo::create` to create a new gizmo, `gizmo::delete` to delete an existing gizmo, `gizmo::search` to find gizmos with certain characteristics, and so on.

40.1 Functions Presented in This Chapter

This chapter discusses the following functions for creating and manipulating hash tables:

- `void Tcl_InitHashTable(Tcl_HashTable *tablePtr,`

  `int keyType)`

Creates a new hash table and stores information about the table at `tablePtr`. `keyType` is either `TCL_STRING_KEYS`, `TCL_ONE_WORD_KEYS`, or an integer greater than 1.

- `Tcl_InitObjHashTable(Tcl_HashTable tablePtr)`

Creates a new hash table and stores information about the table at `*tablePtr`. The keys in the hash table are `Tcl_Obj*` references (cast to `char*`), with key equality determined according to whether the string representations of keys (as returned by `Tcl_GetString`) are the same. Note that this is a front end to `Tcl_InitCustomHashTable`, where the type of the keys is defined to be a managed reference to a `Tcl_Obj`.

- `Tcl_InitCustomHashTable(Tcl_HashTable *tablePtr,`

  `int keyType, Tcl_HashKeyType *typePtr)`

Creates a new hash table and stores information about the table at `tablePtr`. The `keyType` argument should be one of `TCL_CUSTOM_TYPE_KEYS` or `TCL_CUSTOM_PTR_KEYS`, and the actual type of the keys should be more completely described by the `typePtr` argument, which must be a pointer to a static structure containing a key type descriptor. See the reference documentation for a complete description of key type descriptors and how to use this function.

- `void Tcl_DeleteHashTable(Tcl_HashTable *tablePtr)`

Deletes all the entries in the hash table and frees up related storage.

- `Tcl_HashEntry *Tcl_CreateHashEntry(`

  `Tcl_HashTable *tablePtr, char *key, int *newPtr)`

Returns a pointer to the entry in `tablePtr` whose key is `key`, creating a new entry if needed. `newPtr` is set to `1` if a new entry was created or `0` if the entry already existed.

- `Tcl_HashEntry *Tcl_FindHashEntry(Tcl_HashTable *tablePtr,`

  `char *key)`

Returns a pointer to the entry in `tablePtr` whose key is `key`, or `NULL` if no such entry exists.

- `void Tcl_DeleteHashEntry(Tcl_HashEntry *entryPtr)`

Deletes an entry from its hash table.

- `ClientData Tcl_GetHashValue(Tcl_HashEntry *entryPtr)`

Returns the value associated with a hash table entry.

- `void Tcl_SetHashValue(Tcl_HashEntry *entryPtr,`

  `ClientData value)`

Sets the value associated with a hash table entry.

- `char *Tcl_GetHashKey(Tcl_HashTable *tablePtr,`

  `Tcl_HashEntry *entryPtr)`

Returns the key associated with a hash table entry.

- `Tcl_HashEntry *Tcl_FirstHashEntry(Tcl_HashTable *tablePtr,`

  `Tcl_HashSearch *searchPtr)`

Starts a search through all the elements of a hash table. Stores information about the search at `searchPtr` and returns the hash table's first entry or `NULL` if it has no entries.

- `Tcl_HashEntry *Tcl_NextHashEntry(`

  `Tcl_HashSearch *searchPtr)`

Returns the next entry in the search identified by `searchPtr` or `NULL` if all entries in the table have been returned.

- `char *Tcl_HashStats(Tcl_HashTable *tablePtr)`

Returns a string giving usage statistics for `tablePtr`. The string is dynamically allocated and must be freed by the caller.

40.2 Keys and Values

Tcl hash tables support four different kinds of keys. All of the entries in a single hash table must use the same kind of key, but different tables may use different kinds. The most common form of key is a string. In this case each key is a null-terminated string of arbitrary length, such as `gizmo18` or `Waste not want not`. Different entries in a table may have keys of different length. The gizmo implementation uses strings as keys.

The second form of key is a one-word value. In this case each key may be any value that fits in a single word, such as an integer. One-word keys are passed into Tcl using values of type `char *`, so the keys are limited to the size of a character pointer.

The third form of key is an array. In this case each key is an array of integers (C int type). All keys in the table must be the same size.

The last form of key is a reference to a `Tcl_Obj` instance. In this case, each key is a pointer to a `Tcl_Obj`, and the hash table retains a reference to each unique key value. `Tcl_Obj*` keys are passed into the hashing engine by casting them to `char*` and are compared by comparing their string representations (as returned by `Tcl_GetStringFromObj`).

The values for hash table entries are items of type `ClientData`, which are large enough to hold either an integer or a pointer. In most applications, such as the gizmo example, hash table values are pointers to records for objects. These pointers are cast into `ClientData` items when stored in hash table entries, and they are cast back from `ClientData` to object pointers when retrieved from the hash table.

40.3 Creating and Deleting Hash Tables

Each hash table is represented by a C structure of type `Tcl_HashTable`. The client, not Tcl, allocates space for such structures, which are usually global variables or elements of other structures. When calling hash table functions, you provide a pointer to a `Tcl_HashTable` structure as a token for the hash table. You should not use or modify any of the fields of a `Tcl_HashTable` directly. Use the Tcl library functions for this.
Here is how a hash table might be created for the gizmo application:

```
Tcl_HashTable gizmoTable;
...
Tcl_InitHashTable(&gizmoTable, TCL_STRING_KEYS);
```

The first argument to `Tcl_InitHashTable` is a `Tcl_HashTable` pointer, and the second argument is an integer that specifies the sort of keys that will be used for the table. `TCL_STRING_KEYS` means that strings will be used as the keys for the table. If `TCL_ONE_WORD_KEYS` is specified, it means that single-word values such as integers or pointers will be used as keys. If the second argument is neither `TCL_STRING_KEYS` nor `TCL_ONE_WORD_KEYS`, it must be an integer value greater than 1; this means that keys are arrays with the given number of ints in each array. `Tcl_InitHashTable` initializes the structure to refer to an empty hash table with keys as specified. (The additional flag values `TCL_CUSTOM_TYPE_KEYS` and `TCL_CUSTOM_PTR_KEYS` may be used only with `Tcl_InitCustomHashTable`.)
Hash tables whose keys are `Tcl_Obj` references are created with `Tcl_InitObjHashTable`, whose only argument is a `Tcl_HashTable` pointer, just as with the first argument to `Tcl_InitHashTable`.
`Tcl_DeleteHashTable` removes all the entries from a hash table and frees the memory that was allocated for the entries and the table (except space for the `Tcl_HashTable` structure itself, which is the property of the client calling `Tcl_DeleteHashTable`). For example, the following statement could be used to delete the hash table we just initialized:

```
Tcl_DeleteHashTable(&gizmoTable);
```

40.4 Creating Entries

The function Tcl_CreateHashEntry creates an entry with a given key, and Tcl_SetHashValue sets the value associated with the entry. For example, the following code might be used to implement the gizmo::create command, which makes a new gizmo object:

```
int GizmoCreateCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    static unsigned int id = 1;
    int new;
    Tcl_HashEntry *entryPtr;
    Gizmo *gizmoPtr;
    char nameBuf[5 + TCL_INTEGER_SPACE];

    ... check argc, etc. ...

    do {
        sprintf(nameBuf, "gizmo%u", id);
        id++;
        entryPtr = Tcl_CreateHashEntry(&gizmoTable,
                nameBuf, &new);
    } while (!new);

    gizmoPtr = (Gizmo *) ckalloc(sizeof(Gizmo));
    Tcl_SetHashValue(entryPtr, gizmoPtr);

    ... initialize *gizmoPtr, etc. ...

    Tcl_SetResult(interp, nameBuf, TCL_VOLATILE);
    return TCL_OK;
}
```

This code creates a name for the object by concatenating gizmo with the value of the static variable id. It returns the name of the new object as the result of the Tcl command, but during creation it uses a stack-local buffer, nameBuf, to hold the name. GizmoCreateCmd then increments id so that each new object will have a unique name; in thread-safe code this would have to be either protected with a mutex or made into either an interpreter- or thread-local variable. Tcl_CreateHashEntry is called to create a new entry with a key equal to the object's name; it returns a token for the entry. Under normal conditions an entry with the given key will not already exist, in which case

Tcl_CreateHashEntry sets new to 1 to indicate that it created a new entry. However, it is possible for Tcl_CreateHashEntry to be called with a key that already exists in the table. In GizmoCreateCmd this can happen only if a very large number of objects are created, so that id wraps around to zero again. If this happens, Tcl_CreateHashEntry sets new to 0; GizmoCreateCmd will try again with the next larger id until it eventually finds a name that isn't already in use.
After creating the hash table entry, GizmoCreateCmd allocates memory for the object's record and invokes Tcl_SetHashValue to store the record address as the value of the hash table entry. The first argument to Tcl_SetHashValue is a token for a hash table entry, and its second argument, the new value for the entry, can be anything that fits in the space of a ClientData value. After setting the value of the hash table entry, GizmoCreateCmd initializes the new object's record and stores the name of the object (in nameBuf) in the interpreter's result.

Note

Tcl's hash tables restructure themselves as you add entries. A table does not use much memory for the hash buckets when it has only a small number of entries, but it will increase the size of the bucket array as the number of entries increases. Tcl's hash tables operate efficiently even when they have a very large number of entries.

When working with Tcl_Obj keys, the hash table system manages the lifetime of the keys for you. The only constraint is that the object argument to Tcl_CreateHashEntry must be cast to a char* to ensure that the type signature is correctly obeyed. Once you have obtained a Tcl_HashEntry pointer from an object hash table, you use it identically to the one from a string hash table; hash tables do not constrain the types of their values at all.

40.5 Finding Existing Entries

The function Tcl_FindHashEntry locates an existing entry in a hash table. It is similar to Tcl_CreateHashEntry except that it does not create a new entry if the key doesn't exist in the hash table. Tcl_FindHashEntry is typically used to find an object, given its name. For example, the gizmo implementation might contain a utility function called GetGizmo, which is something like Tcl_GetInt except that it translates its string argument to a Gizmo pointer instead of an integer:

```
Gizmo *GetGizmo(Tcl_Interp *interp, char *string) {
    Tcl_HashEntry *entryPtr;
    entryPtr = Tcl_FindHashEntry(&gizmoTable, string);
    if (entryPtr == NULL) {
        Tcl_AppendResult(interp, "no gizmo named \"",
                string, "\"", NULL);
        return NULL;
    }
    return (Gizmo *) Tcl_GetHashValue(entryPtr);
}
```

`GetGizmo` looks up a gizmo name in the gizmo hash table. If the name exists, `GetGizmo` extracts the value from the entry using `Tcl_GetHashValue`, converts it to a `Gizmo` pointer, and returns it. If the name doesn't exist, `GetGizmo` stores an error message in the interpreter's result and returns `NULL`. This is the same when working with a `Tcl_Obj*` hash table, except the key must be cast to `char*` before it is put into `Tcl_FindHashEntry`.
`GetGizmo` can be invoked from any command function that needs to look up a gizmo object. For example, suppose there is a command `gizmo::twist` that performs a "twist" operation on gizmos, and that it takes a gizmo name as its first argument. The command might be implemented like this:

```
int GizmoTwistCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    Gizmo *gizmoPtr;
    ... check argc, etc. ...
    gizmoPtr = GetGizmo(interp, argv[1]);
    if (gizmoPtr == NULL) {
        return TCL_ERROR;
    }
    ... perform twist operation ...
    return TCL_OK;
}
```

40.6 Searching

Tcl provides two functions that you can use to search through all of the entries in a hash table. `Tcl_FirstHashEntry` starts a search and returns the first entry, and `Tcl_NextHashEntry` returns successive entries until the search is complete. For example, suppose you wish to provide a `gizmo::search` command that searches through all existing gizmos and returns a list of the names of the gizmos that meet a certain set of criteria. This command might be implemented as follows:

```
int GizmoSearchCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    Tcl_HashEntry *entryPtr;
    Tcl_HashSearch search;
    Gizmo *gizmoPtr;
    ... process arguments to choose search criteria ...
    for (entryPtr = Tcl_FirstHashEntry(&gizmoTable, &search);
            entryPtr != NULL;
            entryPtr = Tcl_NextHashEntry(&search)) {
        gizmoPtr = (Gizmo *) Tcl_GetHashValue(entryPtr);
        if (...object satisfies search criteria...) {
            Tcl_AppendElement(interp,
                    Tcl_GetHashKey(&gizmoTable, entryPtr));
        }
    }
    return TCL_OK;
}
```

A structure of type `Tcl_HashSearch` is used to keep track of the search; it is possible to carry out multiple searches simultaneously, using a different `Tcl_HashSearch` structure for each. `Tcl_FirstHashEntry` initializes this structure and returns a token for the first entry in the table (or `NULL` if the table is empty). `Tcl_NextHashEntry` uses the information in the structure to step through successive entries in the table; each call to `Tcl_NextHashEntry` returns a pointer to the next entry (in no particular order), and `NULL` is returned when the end of the table is reached. `GizmoSearchCmd` extracts the value from each entry, converts it to a `Gizmo` pointer, and checks whether that object meets the criteria specified in the command's arguments. If so, `GizmoSearchCmd` uses the `Tcl_GetHashKey` function to get the name of the object (i.e., the entry's key) and invokes `Tcl_AppendElement` to append the name to the interpreter's result as a list element. Though the type of the result of `Tcl_GetHashKey` is `char*`, its actual type depends on the key type configured at hash table creation time.

Note

It is not safe to modify the structure of a hash table during a search except to delete the current entry returned by `Tcl_FirstHashEntry` and `Tcl_NextHashEntry`. If you create any entries or delete any entry other than the current one, you should terminate any searches of that table that are in progress.

40.7 Deleting Entries

The function `Tcl_DeleteHashEntry` deletes an entry from a hash table. For example, the following function uses `Tcl_DeleteHashEntry` to implement a `gizmo::delete` command, which takes any number of arguments and deletes the gizmo objects they name:

```
int GizmoDeleteCmd(ClientData clientData,
        Tcl_Interp *interp, int argc, char *argv[]) {
    Tcl_HashEntry *entryPtr;
    Gizmo *gizmoPtr;
    int i;
    for (i = 1; i < argc; i++) {
        entryPtr = Tcl_FindHashEntry(&gizmoTable, argv[i]);
        if (entryPtr == NULL) {
            continue;
        }
        gizmoPtr = (Gizmo *) Tcl_GetHashValue(entryPtr);
        Tcl_DeleteHashEntry(entryPtr);
        ... clean up *gizmoPtr ...
        ckfree((char *) gizmoPtr);
    }
    return TCL_OK;
}
```

`GizmoDeleteCmd` checks each of its arguments to see if it is the name of a gizmo object. If not, the argument is ignored. Otherwise, `GizmoDeleteCmd` extracts a `Gizmo` pointer from the hash table entry and calls `Tcl_DeleteHashEntry` to remove the entry from the hash table. Then it performs internal cleanup on the gizmo object if needed and frees the object's record.

Note

Tcl does not manage the lifetime of the hash values for you. If they are values that need explicit freeing, you should do this when you use `Tcl_DeleteHashEntry` or `Tcl_DeleteHashTable`, or the memory is leaked.

40.8 Statistics

The function `Tcl_HashStats` returns a string containing various statistics about the structure of a hash table. For example, it might be used to implement a `gizmo::stat` command for gizmos:

```
int GizmoStatCmd(ClientData clientData, Tcl_Interp *interp,
        int argc, char *argv[]) {
    if (argc != 1) {
        Tcl_SetResult(interp, "wrong # args", TCL_STATIC);
        return TCL_ERROR;
    }
    Tcl_SetResult(interp, Tcl_HashStats(&gizmoTable), TCL_DYNAMIC);
    return TCL_OK;
}
```

The string returned by `Tcl_HashStats` is dynamically allocated and must be passed to `ckfree` or `Tcl_Free`; `GizmoStatCmd` uses this string to set the interpreter's result using the `Tcl_SetResult` command with the lifetime mode set to `TCL_DYNAMIC`.
The string returned by `Tcl_HashStats` is not formally defined, but it contains human-readable information such as the following:

```
1416 entries in table, 1024 buckets
number of buckets with 0 entries: 60
number of buckets with 1 entries: 591
number of buckets with 2 entries: 302
number of buckets with 3 entries: 67
number of buckets with 4 entries: 5
number of buckets with 5 entries: 0
number of buckets with 6 entries: 0
number of buckets with 7 entries: 0
number of buckets with 8 entries: 0
number of buckets with 9 entries: 0
number of buckets with more than 10 entries: 0
average search distance for entry: 1.4
```

You can use this information to see how efficiently the entries are stored in the hash table. For example, the last line indicates the average number of entries that Tcl will have to check during hash table lookups, assuming that all entries are accessed with equal probability.

41. List and Dictionary Objects

As of version 8.5, Tcl provides two data structures that are easily accessible from both C and Tcl: lists and dictionaries. Manipulating lists at the Tcl scripting level is covered in Chapter 6 and dictionaries in Chapter 7. The current Tcl list API has been around for several years, so you can count on its being present in any reasonably up-to-date Tcl installation. On the other hand, dictionaries (which are based internally on the hash tables discussed in Chapter 40) require Tcl 8.5 or later.

41.1 Functions Presented in This Chapter

This chapter discusses the following functions for manipulating lists and dictionaries:

- `Tcl_Obj *Tcl_NewListObj(int` *`objc`*`, Tcl_Obj *CONST` *`objv`*`[])`

Creates a new list object from an array *`objv`* pointing to Tcl objects and the number of objects to use, *`objc`*.

- `int Tcl_ListObjAppendElement(Tcl_Interp` *`*interp`*`,`

  `Tcl_Obj` *`*listPtr`*`, Tcl_Obj` *`*objPtr`*`)`

Given a pointer *`listPtr`* to an object containing a list, appends *`objPtr`* to the list.

- `int Tcl_ListObjAppendList(Tcl_Interp` *`*interp`*`,`

  `Tcl_Obj` *`*listPtr`*`, Tcl_Obj` *`*elemListPtr`*`)`

Given a pointer *`listPtr`* to an object containing a list (or that can be converted to one), appends a second list *`elemListPtr`* to *`listPtr`*.

- `Tcl_SetListObj(Tcl_Obj` *`*objPtr`*`, int` *`objc`*`,`

  `Tcl_Obj *CONST` *`objv`*`[])`

Sets the list object *`objPtr`* to contain *`objc`* elements contained in *`objv`*.

- `int Tcl_ListObjGetElements(Tcl_Interp` *`*interp`*`,`

  `Tcl_Obj` *`*listPtr`*`, int` *`*objcPtr`*`,`

  `Tcl_Obj` *`***objvPtr`*`)`

Returns a count and a pointer to an array of the elements in a list object.

- `int Tcl_ListObjIndex(Tcl_Interp` *`*interp`*`,`

  `Tcl_Obj` *`*listPtr`*`, int` *`index`*`, Tcl_Obj` *`**objPtrPtr`*`)`

Given a pointer, *`listPtr`*, to an object containing a list, places the object at index *`index`* in *`objPtrPtr`*. If *`index`* is out of range (`-1`, or greater than the index of the last element), `NULL` is stored in *`objPtrPtr`* and `TCL_OK` is returned.

- `int Tcl_ListObjLength(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`listPtr`*`, int *`*`intPtr`*`)`

Given a pointer *`listPtr`* to an object containing a list, places the list's length in *`intPtr`*.

- `int Tcl_ListObjReplace(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`listPtr`*`, int `*`first`*`, int `*`count`*`, int `*`objc`*`,`

  `Tcl_Obj *CONST `*`objv`*`[])`

Deletes *`count`* elements from *`listPtr`*, starting with the element indexed by *`first`*, and then replaces them with *`objc`* number of elements from the *`objv`* array of `Tcl_Obj`s, starting with the first element. If *`objv`* is `NULL`, no new elements are added. If the argument *`first`* is `0` or negative, it refers to the first element. If *`first`* is greater than or equal to the number of elements in the list, no elements are deleted; the new elements are appended to the list. *`count`* gives the number of elements to replace. If *`count`* is `0` or negative, no elements are deleted; the new elements are simply inserted before the one designated by *`first`*.

- `Tcl_Obj *Tcl_NewDictObj()`

Creates a new, empty dictionary object.

- `int Tcl_DictObjPut(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`dictPtr`*`, Tcl_Obj *`*`keyPtr`*`,`

  `Tcl_Obj *`*`valuePtr`*`)`

Adds a key-value pair to a dictionary, or updates the value for a key if that key already has a mapping in the dictionary.

- `int Tcl_DictObjPutKeyList(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`dictPtr`*`, int `*`keyc`*`, const Tcl_Obj *`*`keyv`*`,`

  `Tcl_Obj *`*`valuePtr`*`)`

Adds a key-value pair to a nested dictionary, or updates the value for a key if that key already has a mapping in the dictionary. The *`keyv`* argument specifies a list of keys (with outermost keys first) that acts as a path to the key-value pair to be affected. Nested dictionaries are created for nonterminal keys where they do not already exist.

- `int Tcl_DictObjGet(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`dictPtr`*`, Tcl_Obj *`*`keyPtr`*`,`

  `Tcl_Obj **`*`valuePtrPtr`*`)`

Given a key, gets its value from the dictionary (or `NULL` if the key is not found in the dictionary).

- `int Tcl_DictObjRemove(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`dictPtr`*`, Tcl_Obj *`*`keyPtr`*`)`

Removes the key/value pair with the given key from the dictionary; the key does not need to be present in the dictionary.

- `int Tcl_DictObjRemoveKeyList(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`dictPtr`*`, int `*`keyc`*`, const Tcl_Obj *`*`keyv`*`)`

Removes a key-value pair from a nested dictionary. The *`keyv`* argument specifies a list of keys (with outermost keys first) that acts as a path to the key-value pair to be affected. All nonterminal keys must exist and have dictionaries as their values.

- `int Tcl_DictObjSize(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`dictPtr`*`, int *`*`sizePtr`*`)`

For the given dictionary, stores in the *`sizePtr`* variable a count of the number of key-value pairs.

- `int Tcl_DictObjFirst(Tcl_Interp *`*`interp`*`,`

  `Tcl_Obj *`*`dictPtr`*`, Tcl_DictSearch *`*`searchPtr`*`,`

  `Tcl_Obj **`*`keyPtrPtr`*`, Tcl_Obj **`*`valuePtrPtr`*`,`

  `int *`*`donePtr`*`)`

Commences an iteration across all the key-value pairs in the given dictionary, placing the key and value in the variables pointed to by the *`keyPtrPtr`* and *`valuePtrPtr`* arguments. Locks the dictionary to enable safe iteration over the dictionary. Stores a search token in the variable pointed to by *`searchPtr`*. Stores `0` in the variable pointed to by *`donePtr`* if there are additional key-value pairs to process, or nonzero if the iteration is complete.

- `void Tcl_DictObjNext(Tcl_DictSearch *`*`searchPtr`*`,`

  `Tcl_Obj *`*`keyPtrPtr`*`, Tcl_Obj **`*`valuePtrPtr`*`,`

  `int *`*`donePtr`*`)`

Given a search token, retrieves the next key-value pair in a dictionary, placing the key and value in the variables pointed to by the *`keyPtrPtr`* and *`valuePtrPtr`* arguments. Stores `0` in the variable pointed to by *`donePtr`* if there are additional key-value pairs to process, or nonzero if the iteration is complete.

- `void Tcl_DictObjDone(Tcl_DictSearch *`*`searchPtr`*`)`

Given a search token, terminates a dictionary iteration before you reach the end of the dictionary and unlocks the dictionary. You don’t need to call `Tcl_DictObjDone` if a previous call to `Tcl_DictObjFirst` or `Tcl_DictObjNext` indicates that the iteration has terminated. It is safe to call `Tcl_DictObjDone` multiple times with the same search token.

41.2 Lists

Lists are implemented internally as position-indexed C arrays (rather than linked lists or Tcl arrays), so operations on them are generally very fast. The list C API supports the same basic operations that the Tcl list commands do: creating lists; adding elements to lists; combining lists; getting, setting, and replacing list elements; as well as finding the length of the list.

Like other object types, lists have a function, `Tcl_NewListObj`, that creates a new, empty Tcl object, as in the following example:

```
Tcl_Obj *listobj, strobj, intobj;
listobj = Tcl_NewListObj(0, NULL);
strobj = Tcl_NewStringObj("Answer", -1);
intobj = Tcl_NewIntObj(42);
Tcl_ListObjAppendElement(interp, listobj, strobj);
Tcl_ListObjAppendElement(interp, listobj, intobj);
```

In this example, we first create an empty list. Then we add two elements to it using the `Tcl_ListObjAppendElement` function. It is not necessary to increase the reference count of the `intobj` and `strobj` objects; `Tcl_ListObjAppendElement` takes care of doing that because the objects have at least one reference to them while associated with the list. You can also take one list object and extend it with the elements of another list object by using the `Tcl_ListObjAppendList` function.

Section 30.4 of the chapter on design philosophy discussed the use of a tagged list, such as `max 105 min 89 average 96`, as a means of passing information between Tcl and C. Such a list is useful in and of itself, because it describes the data. Furthermore, it could also be passed to the `array set` command in order to create an array:

```
array set temps {max 105 min 89 average 96}
```

Building this array from C is easy when `Tcl_ListObjAppendElement`, as in the first example, is used to build up the list piece by piece. Another approach is to use an array of `Tcl_Obj` pointers, where each `Tcl_Obj` is intended as an element in the resulting list, and pass the array to `Tcl_NewListObj` along with a count of the number of objects:

```
Tcl_Obj *templist;
Tcl_Obj *objs[6];
min = 89;
max = 105;
average = 96;

objs[0] = Tcl_NewStringObj("max", -1);
objs[1] = Tcl_NewIntObj(max);
objs[2] = Tcl_NewStringObj("min", -1);
objs[3] = Tcl_NewIntObj(min);
objs[4] = Tcl_NewStringObj("average", -1);
objs[5] = Tcl_NewIntObj(average);
templist = Tcl_NewListObj(6, objs);
Tcl_IncrRefCount(templist);
```

It's also possible to start with an object not originally created as a list object and do list operations on it. As long as the string representation of the object is that of a well-formed Tcl list, the accessor functions can convert the object for use with the list functions. If the object does not have a valid list structure, the accessor functions return `TCL_ERROR`. For example, the following code first builds a `Tcl_Obj` based on a string. The subsequent list operations then convert the object to have a list representation:

```
int objnum = 0;
int listlength = 0;
Tcl_Obj *element;
Tcl_Obj *mylist = Tcl_NewStringObj("a b c d", -1);
Tcl_Obj **objs;

Tcl_ListObjIndex(interp, mylist, 1, &element);
Tcl_ListObjGetElements(interp, mylist, &objnum, &objs);
Tcl_ListObjLength(interp, mylist, &listlength);
```

The `Tcl_ListObjIndex` returns an object representing the second element of the list, in this case `b`, and places the result in `element`. `Tcl_ListObjGetElements` retrieves all of the list elements as an array of `Tcl_Obj`s and stores a pointer to the array in `objs` and a count of the elements in `objnum`. The `Tcl_Obj` array pointed to is managed by Tcl and should not be freed or written to by the caller. The final example stores a count of the list elements in the variable `listlength`.

The `Tcl_ListObjReplace` function can perform both element insertion and deletion and so serves as the basis of many list operations. Its arguments in order are the Tcl interpreter, a pointer to a list object to process, the index of the first element to delete, a count of the number of consecutive elements

to delete, the count of the number of elements to insert in their place, and an array of `Tcl_Obj`s that serve as replacement elements. Any element deleted from the list has its `Tcl_Obj` reference count decremented automatically, and any element added to the list has its `Tcl_Obj` reference count incremented automatically.
For example, the following replaces the single element at index 3 with a single new element:

```
Tcl_Obj *newObj = Tcl_NewStringObj("Carol", -1);
Tcl_ListObjReplace(interp, mylist, 2, 1, 1, &newObj);
```

This example, on the other hand, inserts a new element at the beginning of the list:

```
Tcl_Obj *newObj = Tcl_NewStringObj("Dean", -1);
Tcl_ListObjReplace(interp, mylist, 0, 0, 1, &newObj);
```

Naturally, lists can also contain other list objects, which allows us to nest lists within lists, or dictionaries within lists, or lists within dictionaries, and so on. However, be careful not to create a structure that is so complex that it is difficult for someone else to understand how it works, or for you to understand if you have to revisit your program later—if you have a list of lists of dictionaries, perhaps it’s time to review your code!

41.3 Dictionaries

Dictionaries, as we have seen for Tcl, are a mapping of values to unique keys. From the scripting perspective, the values can be any string, which means that they can be interpreted as lists or nested dictionaries. Internally, dictionaries are based on Tcl’s hash tables, discussed previously in Chapter 40, but Tcl’s C API provides a set of functions to deal with dictionaries as objects quite easily.
You can use the `Tcl_NewDictObj` function to create a new, empty dictionary. You can then add key-value mappings to the dictionary with the `Tcl_DictObjPut` function. For example, the previous section described a “tagged list” with a format such as `max 105 min 89 average 96`. Obviously, this is a dictionary format, and we could create it directly as a dictionary as follows:

```
Tcl_Obj *dictObj;
Tcl_Obj *objs[6];
int i;
min = 89;
max = 105;
average = 96;

objs[0] = Tcl_NewStringObj("max", -1);
objs[1] = Tcl_NewIntObj(max);
objs[2] = Tcl_NewStringObj("min", -1);
objs[3] = Tcl_NewIntObj(min);
objs[4] = Tcl_NewStringObj("average", -1);
objs[5] = Tcl_NewIntObj(average);
dictObj = Tcl_NewDictObj();
for (i=0 ; i<6 ; i+=2) {
    Tcl_DictObjPut(interp, dictObj, objs[i], objs[i+1]);
}
```

The `Tcl_DictObjPut` function automatically increments the reference counts for both the key and the value if it proves necessary to store them in the dictionary (i.e., if they didn't already exist in the dictionary).

Note

The dictionary object cannot be shared if you pass it to `Tcl_DictObjPut` or any other function that modifies a dictionary; a `Tcl_Panic` is triggered it if is shared.

The `Tcl_DictObjGet` function retrieves a value associated with a key. For example, the following code accesses the dictionary just created and retrieves a pointer to the object representing the value of the `min` key, storing it in the variable `value`:

```
Tcl_Obj *key, *value;
result int;
key = Tcl_NewStringObj("min", -1);
result = Tcl_DictObjGet(interp, dictPtr, key, &value);
```

The value stored in `value` would be `NULL` if the key didn't exist. If the object referred to by `dictPtr` cannot be converted to a dictionary, the `Tcl_DictObjGet` function returns `TCL_ERROR`.
If you map a key to a string value that has the form of a nested dictionary, that value is not automatically treated as a dictionary. For example, although

the value shown in the following code has the string format of a dictionary, it is still treated as a string:

```
Tcl_Obj *key = Tcl_NewStringObj("NAME", -1);
Tcl_Obj *value = Tcl_NewStringObj(
                    "{FIRST Dean LAST Akamine}", -1);
Tcl_Obj *dictObj = Tcl_NewDictObj();
Tcl_DictObjPut(interp, dictObj, key, value);
```

However, Tcl automatically converts the object to a dictionary representation if it is accessed with a dictionary function. On the other hand, if you know that you're manipulating nested dictionary structures, you can use the `Tcl_DictObjPutKeyList` function, which accepts an array of key objects. For example, here is the implementation of Tcl's `dict set` command function, which allows the user to specify any number of nested keys as arguments, followed by the value to map to the terminal key:

```
static int
DictSetCmd(
    ClientData dummy,
    Tcl_Interp *interp,
    int objc,
    Tcl_Obj *const *objv)
{
    Tcl_Obj *dictPtr, *resultPtr;
    int result, allocatedDict = 0;

    if (objc < 4) {
        Tcl_WrongNumArgs(interp, 1, objv,
                         "varName key ?key ...? value");
        return TCL_ERROR;
    }

    dictPtr = Tcl_ObjGetVar2(interp, objv[1], NULL, 0);
    if (dictPtr == NULL) {
        allocatedDict = 1;
        dictPtr = Tcl_NewDictObj();
    } else if (Tcl_IsShared(dictPtr)) {
        allocatedDict = 1;
        dictPtr = Tcl_DuplicateObj(dictPtr);
    }

    result = Tcl_DictObjPutKeyList(interp, dictPtr, objc-3,
                                   objv+2, objv[objc-1]);
    if (result != TCL_OK) {
        if (allocatedDict) {
            TclDecrRefCount(dictPtr);
        }

        return TCL_ERROR;
    }

    resultPtr = Tcl_ObjSetVar2(interp, objv[1], NULL,
                               dictPtr, TCL_LEAVE_ERR_MSG);
    if (resultPtr == NULL) {
        return TCL_ERROR;
    }
    Tcl_SetObjResult(interp, resultPtr);
    return TCL_OK;
}
```

Note

Notice how the function checks to see if the dictionary object is shared and, if so, duplicates it before modifying it. Otherwise, if the dictionary object were shared, the call to `Tcl_DictObjPutKeyList` would cause a `Tcl_Panic`.

The `Tcl_DictObjSize` function gives you the size of a dictionary (that is, the number of key-value mappings). You can remove a key-value pair from a dictionary using the `Tcl_DictObjRemove` function. This function also decrements the reference count of the objects representing the key and value in the dictionary. It is not an error if the key did not previously exist. For removing key-value pairs from nested dictionary structures, you can call the `Tcl_DictObjRemoveKeyList` function, which accepts an array of keys, as with `Tcl_DictObjPutKeyList`.

Finally, you can iterate over the key-value pairs of a dictionary. The `Tcl_DictObjFirst` function starts a traversal of a particular dictionary, returning the first key-value pair. Subsequent calls to `Tcl_DictObjNext` iterate through the key-value pairs. Each of these functions has a termination indicator to let you know if you've reached the end of the iteration. If you want to stop a search before you reach the end of the dictionary, you must call `Tcl_DictObjDone` to clean up the search state; you don't need to call

`Tcl_DictObjDone` if `Tcl_DictObjNext` reaches the end of the dictionary itself, though it's not an error to do so. Each dictionary traversal has its own unique `Tcl_DictSearch` token associated with it, which must be passed to each of these functions.

Note

If the value of the dictionary is modified during the iteration, either by `Tcl_DictObj` function calls or a function call that causes the object to change to a non-dictionary representation, the iteration is terminated. The next call to `Tcl_DictObjNext` indicates that it has reached the end of the iteration.

As an example of using the dictionary iterators, the following code creates a list consisting of all the values from a dictionary:

```
Tcl_DictSearch search;
Tcl_Obj *key, *value, *valueList;
int done;

/*
 * Assume interp and objPtr are parameters.
 */
if (Tcl_DictObjFirst(interp, objPtr, &search,
        &key, &value, &done) != TCL_OK) {
    return TCL_ERROR;
}

valueList = Tcl_NewListObj(0, NULL);

for (; !done ; Tcl_DictObjNext(&search, &key, &value, &done)) {
    Tcl_ListObjAppendElement(interp, valueList, value);
}
Tcl_DictObjDone(&search);
```

42. Channels

Tcl provides an extremely flexible system for managing input and output based on the concept of *channels*. You are probably familiar with file and TCP socket channels at the Tcl level. They are two of the channel types that Tcl itself provides. The channel system was created in order to provide a flexible, extensible, cross-platform, device-independent means of dealing with input and output. The channel design takes a two-tiered approach: from Tcl, or using a high-level approach in C, all channels are used in more or less the same way. Standard operations such as `puts`, `read`, `chan flush`, and so forth are the same for all channels, whatever lowlevel device (files, TCP sockets, real hardware devices, etc.) implements them. Lower-level drivers implement the specifics of each channel type, creating a bridge between the generic layer and the “device.” This design means that it is possible to create new channel types that behave exactly as built-in channels do.
This chapter explains how to use channels from C and how to write a driver for a new channel type.

42.1 Functions Presented in This Chapter

Tcl provides a rich set of functions for interacting with channels. Most of these functions take a `Tcl_Channel` as an argument or return it as a result.

42.1.1 Basic Channel Operations

The following functions perform basic channel operations such as opening, closing, reading, writing, and so on:

- `Tcl_Channel Tcl_FSOpenFileChannel(Tcl_Interp` *`interp`*`, Tcl_Obj` *`*pathPtr`*`, const char` *`*modeString`*`, int` *`permissions`*`)`

Opens a file specified by *`pathPtr`* and returns a channel handle. The syntax and meaning of all arguments are similar to those given in the Tcl `open` command when opening a file. (Replaces the older string-based `Tcl_OpenFileChannel` function.)

- `Tcl_Channel Tcl_GetStdChannel(int` *`type`*`)`

Returns a channel handle for a standard I/O channel, where *`type`* is one of

TCL_STDIN, TCL_STDOUT, or TCL_STDERR.

- `Tcl_SetStdChannel(Tcl_Channel` *`channel`*`, int` *`type`*`)`

Sets an existing channel to use as a standard I/O channel, where *`type`* is one of `TCL_STDIN`, `TCL_STDOUT`, or `TCL_STDERR`.

- `int Tcl_Close(Tcl_Interp *`*`interp`*`, Tcl_Channel` *`channel`*`)`

Destroys the channel. Buffered output is flushed to the channel's output device prior to destroying the channel, and any buffered input is discarded. The channel should not be registered in any interpreter when `Tcl_Close` is called; call `Tcl_UnregisterChannel` instead.

- `int Tcl_ReadChars(Tcl_Channel` *`channel`*`,`

  `Tcl_Obj *`*`readObjPtr`*`, int` *`charsToRead`*`,`

  `int` *`appendFlag`*`)`

Consumes bytes from *`channel`*, converting them to UTF-8 based on the channel's encoding and storing the produced data in *`readObjPtr`*'s string representation. The return value is the number of characters, up to *`charsToRead`*, that were stored in *`readObjPtr`*. If an error occurs while reading, the return value is `-1`. (Replaces the older `Tcl_Read` function, which doesn't support encoding translation.)

- `int Tcl_ReadRaw(Tcl_Channel` *`channel`*`, char *`*`readBuf`*`,`

  `int` *`bytesToRead`*`)`

Similar to `Tcl_ReadChars`, except used by transformational channel drivers in stacked channel applications. *`bytesToRead`* bytes are read from the channel below and copied to *`readBuf`* without encoding translation or other changes.

- `int Tcl_GetsObj(Tcl_Channel` *`channel`*`, Tcl_Obj *`*`lineObjPtr`*`)`

  `int Tcl_Gets(Tcl_Channel` *`channel`*`, Tcl_DString *`*`lineRead`*`)`

`Tcl_GetsObj` consumes bytes from *`channel`*, converting them to UTF-8 based on the channel's encoding, until a full line of input has been seen. All of the characters of the line except for the terminating end-of-line character(s) are appended to *`lineObjPtr`*'s string representation; the end-of-line character(s) are read and discarded. Returns the number of characters stored in *`lineObjPtr`*, or `-1` in case of an error condition or an end-of-file condition; also returns `-1` and consumes no data on a non-blocking channel if a complete line of data was not available. `Tcl_Gets` is the same as `Tcl_GetsObj` except the resulting characters are appended to the dynamic string given by *`lineRead`* rather than a Tcl object.

- `int Tcl_Ungets(Tcl_Channel` *`channel`*`, const char *`*`input`*`,`

  `int` *`inputLen`*`, int` *`addAtEnd`*`)`

Adds data from *`input`* to the input queue of a channel; *`inputLen`* gives the number of bytes to add. A nonzero value of *`addAtEnd`* indicates that the data is to be added at the end of the queue; otherwise, it is to be added at the head of the queue. `Tcl_Ungets` returns *`inputLen`*, or `-1` if an error occurs.

- `int Tcl_WriteObj(Tcl_Channel` *`channel`*`,`
  `Tcl_Obj *`*`writeObjPtr`*`)`
  `int Tcl_WriteChars(Tcl_Channel` *`channel`*`,`
  `const char *`*`charBuf`*`, int` *`bytesToWrite`*`)`

`Tcl_WriteObj` writes the string representation of *`writeObjPtr`* to the channel, converting the characters to the channel's encoding. `Tcl_WriteChars` does the same, except *`charBuf`* contains the UTF-8 characters; *`bytesToWrite`* specifies the number of bytes to write, or `-1` to indicate that *`charBuf`* is a nullterminated string and all characters should be written. (`Tcl_WriteChars` replaces the older `Tcl_Write` function, which doesn't support encoding translation.) Returns the number of bytes written, or `-1` in case of an error.

- `int Tcl_WriteRaw(Tcl_Channel` *`channel`*`, const char *`*`byteBuf`*`,`
  `int` *`bytesToWrite`*`)`

Similar to `Tcl_WriteChars`, except used by transformational channel drivers in stacked channel applications. *`bytesToWrite`* bytes are read from *`byteBuf`* and written to the channel below without encoding translation or other changes.

- `int Tcl_Eof(Tcl_Channel` *`channel`*`)`

Returns a nonzero value if *`channel`* encountered an end of file during the last input operation.

- `int Tcl_Flush(Tcl_Channel` *`channel`*`)`

Causes all of the buffered output data for *`channel`* to be written to its underlying file or device as soon as possible.

- `int Tcl_InputBlocked(Tcl_Channel` *`channel`*`)`

Returns a nonzero value if *`channel`* is in non-blocking mode and the last input operation returned less data than requested because insufficient data was available. The call always returns `0` if the channel is in blocking mode.

- `int Tcl_InputBuffered(Tcl_Channel` *`channel`*`)`

Returns the number of bytes of input currently buffered in the internal buffers for a channel. If the channel is not open for reading, this function always returns `0`.

- `int Tcl_OutputBuffered(Tcl_Channel` *`channel`*`)`

Returns the number of bytes of output currently buffered in the internal buffers for a channel. If the channel is not open for writing, this function always returns `0`.

- `Tcl_WideInt Tcl_Seek(Tcl_Channel` *`channel`*`,`
  `Tcl_WideInt` *`offset`*`, int` *`seekMode`*`)`

Moves the access point in *`channel`* where subsequent data is read or written. The requested access point is a signed byte *`offset`* relative to the *`seekMode`*, which is one of `SEEK_SET` (start), `SEEK_CUR` (current position), or `SEEK_END` (end).

Buffered output is flushed to the channel, and buffered input is discarded, prior to the seek operation. Returns the new access point, or `-1` in case of an error.

- `Tcl_WideInt Tcl_Tell(Tcl_Channel *channel*)`

Returns the current access point for a channel, or `-1` if the channel does not support seeking.

-
  ```
  int Tcl_TruncateChannel(Tcl_Channel channel,
          Tcl_WideInt length)
  ```

Truncates the file underlying *`channel`* to a given *`length`* of bytes.

-
  ```
  int Tcl_GetChannelOption(Tcl_Interp *interp,
          Tcl_Channel channel, const char *optionName,
          Tcl_DString *optionValue)
  ```

Retrieves the value of a channel option named *`optionName`* and stores the result in *`optionValue`*. If *`optionName`* is `NULL`, the function stores an alternating list of all channel option names and their values in *`optionValue`*. The *`interp`* can be `NULL`.

-
  ```
  int Tcl_SetChannelOption(Tcl_Interp *interp,
          Tcl_Channel channel, const char *optionName,
          const char *newValue)
  ```

Sets *`newValue`* as the value for the channel option given by *`optionName`*. The procedure normally returns `TCL_OK`. If an error occurs, it returns `TCL_ERROR`; in addition, if *`interp`* is non-null, `Tcl_SetChannelOption` leaves an error message in the interpreter's result.

42.1.2 Channel Registration Functions

The following functions manage the registration of channels with interpreters and threads:

-
  ```
  void Tcl_RegisterChannel(Tcl_Interp *interp,
          Tcl_Channel channel)
  ```

Adds a channel to the set of channels accessible in *`interp`*. After this call, Tcl programs executing in that interpreter can refer to the channel in input or output operations using the name given in the call to `Tcl_CreateChannel`. The *`interp`* argument may be `NULL` to add a reference to the channel independent of any interpreter.

-
  ```
  int Tcl_UnregisterChannel(Tcl_Interp *interp,
          Tcl_Channel channel)
    int Tcl_DetachChannel(Tcl_Interp *interp,
          Tcl_Channel channel)
  ```

Removes a channel from the set of channels accessible in *`interp`*. After this call, Tcl programs can no longer use the channel's name to refer to the channel in that interpreter. The *interp* argument may be NULL to remove a reference to the channel independent of any interpreter. In the case of Tcl_UnregisterChannel, if this operation removes the last registration of the channel in any interpreter, the channel is also closed and destroyed.

- `int Tcl_IsChannelShared(Tcl_Channel channel)`

Returns 1 if *channel* is shared among multiple interpreters, otherwise 0.

- `int Tcl_IsChannelRegistered(Tcl_Interp *interp,`

  `Tcl_Channel channel)`

  `void Tcl_CutChannel(Tcl_Channel channel)`

Removes the specified channel from the list of all channels belonging to the current thread (if threads are present). The operation may not be performed on channels registered with an interpreter.

- `void Tcl_SpliceChannel(Tcl_Channel channel)`

Adds the specified channel to the list of all channels belonging to the current thread (if threads are present). The operation may not be performed on channels registered with an interpreter. The channel must have been previously cut from a thread with Tcl_CutThread.

42.1.3 Channel Attribute Functions

The following functions allow you to query or set attributes of a particular channel:

- `int Tcl_IsChannelExisting(CONST char *channelName)`

Returns 1 if a channel with the given name exists, 0 otherwise.

- `int Tcl_IsStandardChannel(Tcl_Channel channel)`

Returns 1 if the channel is one of the three standard channels—stdin, stdout, or stderr—0 otherwise.

- `ClientData Tcl_GetChannelInstanceData(Tcl_Channel channel)`

Fetches the instance data for a given channel.

- `Tcl_ChannelType *Tcl_GetChannelType(Tcl_Channel channel)`

Fetches a pointer to the channel's type.

- `CONST char *Tcl_GetChannelName(Tcl_Channel channel)`

Returns the name of a given channel.

- `int Tcl_GetChannelHandle(Tcl_Channel channel,`

  `int direction, ClientData handlePtr)`

Places the OS-specific device handle (such as a *FILE **) associated with a channel and direction (TCL_READABLE or TCL_WRITABLE) in *handlePtr*. Returns TCL_ERROR if there is no handle.

- `Tcl_ThreadId Tcl_GetChannelThread(Tcl_Channel *channel*)`

Returns the ID of the thread currently managing the given channel.

- `int Tcl_GetChannelMode(Tcl_Channel *channel*)`

Returns an ORed combination of `TCL_READABLE` and `TCL_WRITABLE`.

- `int Tcl_GetChannelBufferSize(Tcl_Channel *channel*)`

Returns the channel's buffer size, in bytes.

- `Tcl_SetChannelBufferSize(Tcl_Channel *channel*, int *size*)`

Sets the channel's buffer size, in bytes.

-
  ```
  int Tcl_IsChannelRegistered(Tcl_Interp interp,
          Tcl_Channel channel)
  ```

Returns `1` if *`channel`* has been registered in interpreter *`interp`*, otherwise `0`.

42.1.4 Channel Query Functions

The following functions provide tools for retrieving information about channels in an interpreter:

-
  ```
  Tcl_Channel Tcl_GetChannel(Tcl_Interp interp,
          const char *channelName, int *modePtr)
  ```

Given the name of a channel handle in Tcl, returns the `Tcl_Channel` associated with it and the *`modePtr`*, which is an integer ORed combination of `TCL_READABLE` and `TCL_WRITABLE`.

-
  ```
  int Tcl_GetChannelNames(Tcl_Interp interp)
   int Tcl_GetChannelNamesEx(Tcl_Interp interp,
          const char *pattern)
  ```

Writes the names of the registered channels to the interpreter's result as a list object. `Tcl_GetChannelNamesEx` filters these names according to the *`pattern`* using the same syntax as `string match`.

42.1.5 Channel Type Definition Functions

The following functions are related to creating and using custom channel types:

-
  ```
  Tcl_Channel Tcl_CreateChannel(
          Tcl_ChannelType *typePtr, CONST char *channelName,
          ClientData instanceData, int mask)
  ```

Creates a new channel of type *`typePtr`* with name *`channelName`*.

-
  ```
  Tcl_Channel Tcl_StackChannel(Tcl_Interp interp,
          Tcl_ChannelType *typePtr, ClientData instanceData,
  ```

`int mask, Tcl_Channel channel)`

Stacks a new channel on an existing `channel` with the same name that was registered for `channel` by `Tcl_RegisterChannel`. The `mask` parameter specifies the operations that are allowed on the new channel. These can be a subset of the operations allowed on the original channel. Other options are as for `Tcl_CreateChannel`.

- `int Tcl_UnstackChannel(Tcl_Interp interp,`
  `Tcl_Channel channel)`

Reverses the process of stacking a channel. The old channel is associated with the channel name, and the processing module added by `Tcl_StackChannel` is destroyed. If there is no old channel, `Tcl_UnstackChannel` is equivalent to `Tcl_Close`.

- `Tcl_Channel Tcl_GetStackedChannel(Tcl_Channel channel)`

Returns the channel in the stack of channels that is just below the supplied `channel`.

- `Tcl_Channel Tcl_GetTopChannel(Tcl_Channel channel)`

Returns the top channel in the stack of channels of which the supplied `channel` is a part.

- `int Tcl_BadChannelOption(Tcl_Interp *interp,`
  `CONST char *optionName, CONST char *optionList)`

Called from the channel option get/set functions to indicate that `optionName` is not a valid option for this channel.

- `Tcl_NotifyChannel(Tcl_Channel channel, int mask)`

Called by the channel driver to indicate to the generic layer that the events specified by `mask` (an OR'ed combination of `TCL_READABLE`, `TCL_WRITABLE`, and `TCL_EXCEPTION`) have occurred.

- `void Tcl_ClearChannelHandlers(Tcl_Channel channel)`

Removes all channel handlers and event scripts associated with the specified channel.

- `int Tcl_ChannelBuffered(Tcl_Channel channel)`

Returns the number, in bytes, currently in the channel's input buffer.

42.2 Channel Operations

Tcl provides a rich set of operations, described in Section 42.1.1, for interacting with channels. Most of these functions take a `Tcl_Channel` structure as an argument or return it as a result. The following code shows the definition of a command function implementing a `filetovar` command:

```
static int
FileToVarCmd(ClientData clientData, Tcl_Interp *interp,
             int objc, Tcl_Obj *CONST objv[]) {
    Tcl_Channel chan;
    Tcl_Obj *buffer;

    if (objc != 3) {
        Tcl_WrongNumArgs(interp, 1, objv, "file varname");
        return TCL_ERROR;
    }

    chan = Tcl_FSOpenFileChannel(interp, objv[1], "r", 0);
    if (chan == NULL) {
        return TCL_ERROR;
    }

    buffer = Tcl_NewObj();
    if (Tcl_ReadChars(chan, buffer, -1, 0) < 0) {
        Tcl_Close(interp, chan);
        return TCL_ERROR;
    }
    Tcl_Close(interp, chan);

    Tcl_SetVar2Ex(interp, Tcl_GetString(objv[2]),
                  NULL, buffer, 0);

    return TCL_OK;
}
```

The idea is that executing the following from a script

```
filetovar /tmp/somefile content
```

has the same effect as

```
set fid [open /tmp/somefile]
set content [read $fid]
close $fid
```

The command procedure expects two arguments the name of a file to read and the name of a variable in which to store the content. The `Tcl_FSOpenFileChannel` function takes a Tcl object containing the file name to open and a mode string that is interpreted in the same way as the mode argument to Tcl's `open` command. It returns a `Tcl_Channel` object that can then be passed to other channel I/O commands.
The actual read is taken care of by the `Tcl_ReadChars` command, which takes as

arguments a channel, a Tcl object in which to place the data, the length of data to read, and a flag indicating whether the data should be appended or not. In this case, the length argument is `-1`, which tells `Tcl_ReadChars` to read the entire file. This function is similar to the Tcl `read` command. `Tcl_Close` then closes the file, and the content is assigned as the value of the variable using `Tcl_SetVar2Ex`.

As you can see, most of the channel functions used in this example are analogous to Tcl script-level commands. Likewise, most of the Tcl I/O commands have roughly equivalent C functions, as shown in Table 42.1. Especially useful are the `Tcl_GetChannelOption` and `Tcl_SetChannelOption` commands, which let us get and set channel options. The following sets the buffer size for a channel:

```
Tcl_SetChannelOption(interp, chan, "-buffersize", "10000");
```

Table 42.1 Tcl Channel Commands and Analogous C Functions

Tcl command	Analogous C functions
`open`	`Tcl_FSOpenFileChannel` for files
`exec`, `"open |"`	`Tcl_OpenCommandChannel` (discussed in further detail in Chapter 45)
`close`	`Tcl_Close`
`puts`	`Tcl_WriteObj`, `Tcl_WriteChars`, `Tcl_WriteRaw`
`read`	`Tcl_ReadChars`, `Tcl_ReadRaw`
`gets`	`Tcl_Gets`, `Tcl_GetsObj`
`chan eof`	`Tcl_Eof`
`chan blocked`	`Tcl_InputBlocked`
`chan flush`	`Tcl_Flush`
`chan seek`	`Tcl_Seek`
`chan tell`	`Tcl_Tell`
`chan truncate`	`Tcl_TruncateChannel`
`chan configure`	`Tcl_GetChannelOption`, `Tcl_SetChannelOption`
`chan pending`	`Tcl_InputBuffered`, `Tcl_OutputBuffered`

The next example retrieves the address, host name, and port number on a socket channel and stores the results into a `DString`:

```
Tcl_DString optionValue;
Tcl_DStringInit(optionValue);
Tcl_GetChannelOption(interp, chan, "-sockname", &optionValue);
```

There are also several functions for querying attributes of the channel, as listed in Section 42.1.3.

42.3 Registering Channels

If you create a channel in Tcl (e.g., with `open`), a channel identifier is returned that lets you manipulate the channel from the scripting level. On the other hand, if you create a channel using Tcl's C API, the channel is not automatically exposed at the scripting level. You would have to do all channel interaction at the C level with `Tcl_Channel`.
If you want to expose a channel to the scripting level, you must call the `Tcl_RegisterChannel` function to register it in each interpreter in which you want to access it. You can register a channel in as many interpreters as you like, as long as they are in the same thread.
The following code implements a `randomfile` package. It opens the Unix `/dev/random` file as a source of random data, sets the channel encoding to binary, and registers the channel with the Tcl interpreter. Then it retrieves the name of the channel created and assigns it to the variable `random`:

```
#include <tcl.h>

int Randomfile_Init(Tcl_Interp *interp) {
    Tcl_Channel chan;
    Tcl_Obj *filename;
#ifdef USE_TCL_STUBS
    if (Tcl_InitStubs(interp, "8", 0) == NULL) {
        return TCL_ERROR;
    }
#endif

    filename = Tcl_NewStringObj("/dev/random", -1);
    Tcl_IncrRefCount(filename);
    chan = Tcl_FSOpenFileChannel(interp, filename, "r", 0);
    Tcl_DecrRefCount(filename);

    if (chan == NULL) {
        return TCL_ERROR;
    }
    Tcl_SetChannelOption(interp, chan, "-encoding",
                         "binary");
    Tcl_RegisterChannel(interp, chan);
    Tcl_SetVar(interp, "random",
               Tcl_GetChannelName(chan), 0);

    if ( Tcl_PkgProvide(interp,
                        "randomfile",
                        "0.1") != TCL_OK ) {
        return TCL_ERROR;
    }
    return TCL_OK;
}
```

Once you have compiled this code and loaded it into a Tcl script, you could read 10 bytes of random data with a command like

```
read $random 10
```

Once you have registered a channel with Tcl, you should not close it from the C level with `Tcl_Close`. The proper way to remove a channel from the Tcl interpreter is with the `Tcl_UnregisterChannel` function, which makes the channel invisible to the specified interpreter and, if there are no more references to it, closes it. In contrast, `Tcl_DetachChannel` removes the channel from an interpreter but does not attempt to close it.
Although you can share a channel with multiple interpreters, all of those interpreters must be in the same thread. You cannot share channels across

threads. However, you can move a channel from one thread to another. To do so, the channel must not be registered with any interpreters; call `Tcl_DetachChannel` if necessary. Then you must “cut” the channel from its current thread with the `Tcl_CutChannel` function. After the channel is no longer associated with a particular thread, you can then call `Tcl_SpliceChannel` from a thread to associate the channel with that thread. See Chapter 46 for more information about threaded Tcl programming.
You can also get a `Tcl_Channel` handle for a channel even if it was created at the scripting level. Given the name of a channel, such as `file1`, the `Tcl_GetChannel` function returns a `Tcl_Channel` for it:

```
Tcl_Channel chan;
int mode = 0;
chan = Tcl_GetChannel(interp, "file1", &mode);
```

42.4 Standard Channels

You are probably already familiar with using Tcl’s standard channels: `stdin`, `stdout`, and `stderr` (covered in Chapter 11). It is possible to manipulate them from C, which is useful if you want to replace one of the standard channels with one of your own creation.
The `Tcl_GetStdChannel` function returns a channel handle for a standard I/O channel. The only argument is one of `TCL_STDIN`, `TCL_STDOUT`, or `TCL_STDERR`. You can also use an existing channel as a standard I/O channel with the `Tcl_SetStdChannel` function:

```
Tcl_SetStdChannel(myChan, TCL_STDOUT);
Tcl_RegisterChannel(NULL, myChan);
```

Note

The call to `Tcl_RegisterChannel` after `Tcl_SetStdChannel` is required in current versions of Tcl. See the reference documentation for more information.

As an example of how this feature is used, Apache Rivet creates a special Apache channel that uses the Apache web server’s API to send data through the server to the browser. Rivet sets this channel as the standard output channel, so that regular `puts` commands, such as `puts "Hello, World"`, are redirected to the Apache channel instead of the normal standard output. This means that ordinary Tcl scripts can be run inside Apache without modification.

Note

If one of the standard channels is set to `NULL`, either by calling `Tcl_SetStdChannel` with a `NULL` channel argument or by calling `Tcl_Close` on the channel, then the next call to `Tcl_CreateChannel` automatically sets the standard channel with the newly created channel. If more than one standard channel is `NULL`, the standard channels are assigned starting with standard input, followed by standard output, and standard error last.

42.5 Creating a New Channel Type

As stated earlier, Tcl provides a complete API for the creation of new channel types. Examples of extensions that implement new channel types are UDP sockets; the `memchan` memory channel implementation, which lets you read and write to memory as if it were a file; and Rivet's Apache channel, which uses the Apache web server's C API to send data to the browser.
The central concept in creating a new driver type is to implement the functions that carry out the different tasks a driver performs, such as reading, writing, flushing, and so on. The basic structure that represents a channel type is `Tcl_ChannelType`:

```
typedef struct Tcl_ChannelType {
    char *typeName;
    Tcl_ChannelTypeVersion version;
    Tcl_DriverCloseProc *closeProc;
    Tcl_DriverInputProc *inputProc;
    Tcl_DriverOutputProc *outputProc;
    Tcl_DriverSeekProc *seekProc;
    Tcl_DriverSetOptionProc *setOptionProc;
    Tcl_DriverGetOptionProc *getOptionProc;
    Tcl_DriverWatchProc *watchProc;
    Tcl_DriverGetHandleProc *getHandleProc;
    Tcl_DriverClose2Proc *close2Proc;
    Tcl_DriverBlockModeProc *blockModeProc;
    Tcl_DriverFlushProc *flushProc;
    Tcl_DriverHandlerProc *handlerProc;
    Tcl_DriverWideSeekProc *wideSeekProc;
} Tcl_ChannelType;
```

Each entry is a pointer to a function that carries out a particular operation. You don't need to provide a function for each channel operation; you can set several of the fields to `NULL` if they don't apply, or implement a function that returns `EINVAL` when called to indicate that the operation is not meaningful on the channel. See the reference documentation for a complete description of the function signatures, their purposes, and which ones are required. We'll see an example of a custom channel type later in Section 42.5.3.

42.5.1 Creating a Custom Channel Instance

After you have implemented your channel type and defined its channel operation functions, you can create instances of the channel type with the `Tcl_CreateChannel` function:

```
Tcl_Channel Tcl_CreateChannel(
        Tcl_ChannelType *typePtr, CONST char *channelName,
        ClientData instanceData, int mask)
```

The *`typePtr`* is a pointer to a previously defined `Tcl_ChannelType` structure. The name used by *`channelName`* must be unique (such as `file1`, `file2`, `sock7`, `sock8`, and so on). The *`instanceData`* variable is your chance to pass some instance-specific data to the channel implementation. Last, the *`mask`* instructs Tcl what operations to allow on the channel, using the following flags:

- `TCL_WRITABLE`—Writing is allowed.
- `TCL_READABLE`—Reading is allowed.

Once you register a channel type with Tcl by creating a channel of that type, the channel type cannot be removed.

42.5.2 Stacked Channels

In Tcl, not only is it possible to create new drivers for specific "devices," it is also possible to create *stacked* channels, which do not directly talk with a device but act as intermediary filter layers between Tcl and some other channel. `TclTLS`, the Tcl interface to OpenSSL, works this way. It doesn't actually create the socket that talks with the web server but adds an SSL filter between the socket and Tcl.
To add a channel to a stack, use `Tcl_StackChannel`:

```
Tcl_Channel Tcl_StackChannel(Tcl_Interp interp,
        Tcl_ChannelType *typePtr, ClientData instanceData,
        int mask, Tcl_Channel channel)
```

Aside from the ever-present interpreter, the function takes a pointer to a `Tcl_ChannelType` structure, such as that illustrated in the preceding example, instance data, a mask describing the mode (`TCL_READABLE`, `TCL_WRITABLE`, or `TCL_EXCEPTION`), and the channel to layer on top of. The newly created top channel structure is returned.
Once the channel is stacked, if any function tries to perform I/O on the original channel, such as with `Tcl_ReadChars` or `Tcl_WriteChars`, the system automatically redirects such calls to the channel on top of the stack. In other words, all `Tcl_Channel` tokens stay valid, independently of where they are in a stack, but there is no "back door" access through these standard I/O functions. At the scripting level, Tcl automatically reassigns the symbolic channel identifier to refer to the top channel in the stack.
It is possible to unstack a channel with `Tcl_UnstackChannel`. This is useful in the case where you wish to remove a filter—perhaps you no longer want to encrypt output to a file. It has a simple prototype:

```
int Tcl_UnstackChannel(Tcl_Interp interp,
        Tcl_Channel channel)
```

The old channel is associated with the symbolic channel name, and the processing module added by `Tcl_StackChannel` is destroyed. If *`channel`* refers to an unstacked channel, `Tcl_UnstackChannel` is equivalent to `Tcl_Close`.

42.5.3 ROT13 Channel

As an example of creating custom channel types, this section describes a stacked channel type that implements the “ROT13” cipher. This is an ancient method of encrypting messages. It works by taking a character, “rotating” 13 places down the alphabet, and using that letter in its place. For instance, the letter `a` becomes `n`, `b` becomes `o`, and so on. Using 13 as the number to shift each letter also has the advantage that performing a second ROT13 transformation “deciphers” the text. Of course, ROT13 is not in any way, shape, or form secure in this day and age, but it makes for a simple example that gives you an idea of what can be done with channels.

Since we are working with a filter, rather than an “endpoint” driver that talks with some device or library external to Tcl, we need a means of attaching our channel to a Tcl channel that is already open. To accomplish this, we create a command, `addrot`, which takes a channel as an argument and, optionally, a number to use in place of 13. In a Tcl script, it looks like this:

```
set fl [open /tmp/outfile w]
addrot $fl
```

Here is the command function to implement the `addrot` command:

```
/* This structure serves as the instance data that gets
 * passed to the channel driver functions. */

typedef struct {
    Tcl_Channel self;
    Tcl_Channel parent;
    int rotate;
} RotInstance;

static int
AddRotCmd(ClientData clientData, Tcl_Interp *interp,
      int objc, Tcl_Obj *CONST objv[]) {
    Tcl_Channel parent;
    int modePtr = 0;
    int rotate = 0;
    RotInstance *rotinstance;

    if (objc < 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "chan ?rotation?");
        return TCL_ERROR;
    }
    /* Get the channel from its name, fail if it's
     * not a valid channel. */
    parent = Tcl_GetChannel(interp, Tcl_GetString(objv[1]),
                            &modePtr);
    if (parent == NULL) {
        Tcl_AppendResult(interp, Tcl_GetString(objv[1]),
                " is not a valid channel", NULL);
    return TCL_ERROR;
    }

    /* We take an additional argument that is the amount
     * to 'rotate' for our cipher. */
    if (objc == 3) {
        if (Tcl_GetIntFromObj(
                interp, objv[2], &rotate) != TCL_OK) {
                return TCL_ERROR;
        }
    } else {
        rotate = 13;
    }
    rotinstance =
        (RotInstance *)Tcl_Alloc(sizeof(RotInstance));

    rotinstance->rotate = rotate;
    rotinstance->parent = parent;

    /* Stack the channel. */
    rotinstance->self = Tcl_StackChannel(
        interp, &RotChan, (ClientData)rotinstance,
        modePtr, parent);

    return TCL_OK;
}
```

The call to `Tcl_GetChannel` returns a `Tcl_Channel` structure for the channel identifier of the existing channel. This channel is the “real” channel that actually sends or pulls data from a file, the channel that our stacked channel reads from or writes to. The code then fills in the fields of a `RotInstance` structure that stores the number of characters to rotate for this instance of the channel and the handle to the original channel. Finally, we use `Tcl_StackChannel` to stack the channel, passing the `RotInstance` structure as instance data. This is where the new channel is created and layered on top of the existing channel. If we were creating a driver for a special device, we would call `Tcl_CreateChannel` in this command instead of `Tcl_StackChannel`.
The key to understanding the new channel’s capabilities is in the `RotChan` structure, which is declared in this way:

```
static
Tcl_ChannelType RotChan = {
    "rot13_channel",
    TCL_CHANNEL_VERSION_3,
    RotcloseProc,
    RotinputProc,
    RotoutputProc,
    NULL,
    RotsetOptionProc,
    RotgetOptionProc,
    RotwatchProc,
    RotgetHandleProc,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL
};
```

Our channel filter is not particularly complex. `RotcloseProc` does nothing more than call `Tcl_Free` to free the storage allocated for the instance data. `RotwatchProc` does nothing, and `RotgetHandleProc` returns an error, because there is no lowlevel handle to return.
We'll examine the output and input functions in detail. In order to define what we're doing clearly, output is when Tcl is sending data out, such as with the `puts` command, and input is when data is being read, such as with `read` or `gets`.

```
static int
RotoutputProc(ClientData instanceData, CONST84 char *buf,
          int toWrite, int *errorCodePtr) {
    RotInstance *rotinstance = (RotInstance *)instanceData;
    char *outbuf;

    outbuf = Tcl_Alloc(toWrite);
    rot13(buf, outbuf, toWrite, rotinstance->rotate);

    /* We have pushed the data down to the next layer
     * of the stack. */

    Tcl_WriteRaw(rotinstance->parent, outbuf, toWrite);
    Tcl_Free(outbuf);
    return toWrite;
}
```

Output functions take four arguments: instance data, a buffer containing data to output, how many bytes the buffer holds, and a pointer to an integer that

can hold a POSIX error code should a problem occur.
`RotoutputProc` allocates a second buffer to contain the transformed text and performs the `rot13` operation, which fills the `outbuf` buffer with the encoded data. The tricky part is the call to `Tcl_WriteRaw`. You *must* use this write function with stacked channels instead of a function like `Tcl_WriteChars`. We do this because we are acting as a filter and must therefore write to the next channel down the stack, which we saved in the instance data when it was created. On the other hand, Tcl automatically redirects calls to functions like `Tcl_WriteChars` to the top channel in a stacked channel.
Were this not a stacked channel (i.e., not a filter), instead of writing to another layer of Tcl, we would write to a specific device. For instance, Apache Rivet's output channel sends the data from the web server to the browser by calling the Apache API—`ap_rwrite(buf, toWrite, globals->r)`—which hands off the buffer to Apache. In the preceding code, after freeing the memory associated with the output buffer, we also return how much data was actually written. In our case, this always equals the amount of data we were asked to write, so we can simply return that value. Especially in the case of non-blocking channels, these numbers are a bit trickier; see the reference documentation for details.
The input function is activated when we read from a channel; for instance:

```
set fl [open /tmp/outfile]
addrot $fl
set data [read $fl]
```

During the read, the Tcl library calls this function to receive data from our channel:

```
static int
RotinputProc(ClientData instanceData, char *buf,
        int bufSize, int *errorCodePtr) {
    RotInstance *rotinstance = (RotInstance *)instanceData;
    char *inbuf;
    int read = 0;

    inbuf = Tcl_Alloc(bufSize);

    /* We need to read the data from the next
     * layer of the stack. */

    read = Tcl_ReadRaw(rotinstance->parent, inbuf, bufSize);
    rot13(inbuf, buf, read, rotinstance->rotate);
    Tcl_Free(inbuf);
    return read;
}
```

Input functions have four variables passed to them: instance data; a buffer to write data to (remember that Tcl wants to read data from this function); the amount of data the buffer can hold; and, again, a pointer to an `int` where we can store an error code. For our channel, the functionality of the input implementation is the reverse of the output function. Because it is a simple filter, we don't have the data ourselves and have to fetch it from the next channel over, via `Tcl_ReadRaw`, noting how many characters were actually read. Analogous to the case of writing, we *must* use this read function with stacked channels instead of a function like `Tcl_ReadChars`, which Tcl automatically redirects to the top channel in a stacked channel. We then "decipher" the data with the `rot13` function into the input buffer `buf`, free the allocated input buffer, and let Tcl know how many bytes are waiting to be collected in `buf` by returning the number we read from the next channel over. Since we're only performing filtering operations, we don't have to do anything with the file itself.

Given that channels are an I/O system, the input and output functions are really the heart of what must be written. Tcl, via `chan configure`, also provides a means to configure options on channels. We can use this to give users a way to change the "rotate" number via a `Tcl_DriverSetOptionProc` function. The Tcl code

```
puts [fconfigure $fl -rotate 10]
```

eventually causes the following function to be called. As arguments, it takes an instance data pointer, an interpreter, the name of the option to set (`-rotate`), and the string value of the option (`10`).

```
static int
RotsetOptionProc(ClientData instanceData, Tcl_Interp *interp,
        CONST char *optionName, CONST char *newValue) {
    RotInstance *rotinstance = (RotInstance *)instanceData;

    if (strcmp(optionName, "-rotate") == 0) {
        if (Tcl_GetInt(interp, newValue,
                       &(rotinstance->rotate)) != TCL_OK) {
            return TCL_ERROR;
        }
    return TCL_OK;
    }
    return Tcl_BadChannelOption(interp, optionName,
                                "rotate");
}
```

As there is only one possible option name, we check for it and attempt to set the value accordingly. If we are given a bad option name, `Tcl_BadChannelOption` is used to return an error with the incorrect name in the `optionName` parameter and a list of the possible names, without the leading hyphen.
There is also a corresponding function to fetch information about current options. It can either return information about one option in particular, such as `-rotate`, or it can return information about all options. From Tcl the two cases look like this:

```
set rotate [chan configure $fl -rotate]
set alloptions [chan configure $fl]
```

The function to get options takes the same arguments as the option set function, with the exception of a `Tcl_DString` where the result is appended. A `NULL` value for `optionName` is a request for all possible options and their values.

```
static int
RotgetOptionProc(ClientData instanceData, Tcl_Interp *interp,
        CONST char *optionName, Tcl_DString *optionValue) {
    RotInstance *rotinstance = (RotInstance *)instanceData;
    Tcl_Obj *rotval;

    if (optionName != NULL &&
            strcmp(optionName, "-rotate") != 0) {
    return Tcl_BadChannelOption(interp, optionName,
                                "rotate");
    }
    rotval = Tcl_NewIntObj(rotinstance->rotate);
    if (optionName == NULL) {
        Tcl_DStringAppendElement(optionValue, "-rotate");
    }
    Tcl_DStringAppendElement(optionValue,
                        Tcl_GetString(rotval));
    return TCL_OK;

}
```

When dealing with a stacked channel, we have to consider that not only is our rotate option visible, but also all the options of the underlying channel or channels, in this case the file channel. In the preceding code, we first check to make sure that the request is valid and returns an error via `Tcl_BadChannelOption` if it is not. We then append the results to the `DString`—just the rotate value if `-rotate` was specified to `chan configure`, otherwise both the string `-rotate` and the value.

Finally, we have a simple implementation of the ROT13 algorithm itself:

```
static void
rot13(char *in, char *out, int len, int rot) {
    int i;
    int inc;
    int outc;

    for (i = 0; i < len; i ++) {
        inc = in[i];
        if (inc >= 65 && inc <= 90) { /* A-Z */
            if (inc + rot > 90) {
                outc = 64 + ((inc + rot) - 90);
            } else {
                outc = inc + rot;
            }
        } else if (inc >= 97 && inc <= 122) { /* a-z */
            if (inc + rot > 122) {
                outc = 96 + ((inc + rot) - 122);
            } else {
                outc = inc + rot;
            }
        } else { /* Other characters passed through. */
            outc = inc;
        }
        out[i] = outc;
    }
}
```

43. Handling Events

This chapter describes Tcl's library functions for event handling. The code you will write for event handling is divided into three parts. The first part consists of code that creates event handlers: it informs Tcl that certain callback functions should be invoked when particular events occur. The second part consists of the callbacks themselves. The third part consists of top-level code that invokes the Tcl event dispatcher to process events.

Tcl supports two principal kinds of events: file events and timer events. Tcl also allows you to create *idle callbacks*, which cause functions to be invoked when Tcl runs out of other things to do; idle callbacks are used in Tk to defer redisplays and other time-consuming computations until all pending events have been processed.

43.1 Functions Presented in This Chapter

Tcl's functions for event handling are the following:

- `void Tcl_CreateChannelHandler(Tcl_Channel` *`channel`*`,`
  `int` *`mask`*`, Tcl_FileProc *`*`callback`*`,`
  `ClientData` *`clientData`*`)`

Arranges for `callback` to be invoked whenever one of the conditions indicated by `mask` occurs for the channel whose handle is `channel`.

- `void Tcl_DeleteChannelHandler(Tcl_Channel` *`channel`*`,`
  `Tcl_FileProc *`*`callback`*`, ClientData` *`clientData`*`)`

Deletes the handler for `channel` that matches a previous invocation of `Tcl_CreateChannelHandler` with the given `callback` and `clientData` arguments.

- `Tcl_TimerToken Tcl_CreateTimerHandler(int` *`milliseconds`*`,`
  `Tcl_TimerProc *`*`callback`*`, ClientData` *`clientData`*`)`

Arranges for `callback` to be invoked after `milliseconds` have elapsed. Returns a token that can be used to cancel the callback.

- `void Tcl_DeleteTimerHandler(Tcl_TimerToken` *`token`*`)`

Cancels the timer callback indicated by `token`, if it has not yet triggered.

- `void Tcl_DoWhenIdle(Tcl_IdleProc *`*`callback`*`,`
  `ClientData` *`clientData`*`)`

Arranges for `callback` to be invoked when Tcl has nothing else to do.

- `void Tcl_CancelIdleCall(Tcl_IdleProc *`*`callback`*`,`

```
                    ClientData clientData)
```
Deletes any existing idle callbacks for `callback` and `clientData`.

- `void Tcl_DoOneEvent(int flags)`

Processes a single event of any sort and then returns. `flags` is normally `0` but may be used to restrict the events that are processed or to return immediately if there are no pending events.

- `void Tcl_SetMainLoop(Tcl_MainLoopProc *mainLoopProc)`

Installs a main-loop function into Tcl that will be called by `Tcl_Main` after the processing of the application startup code (including the script specified on the command line).

- `void Tk_MainLoop(void)`

Convenience function that processes events until every window created by this thread has been destroyed. Available only when Tk is present.

43.2 Channel Events

Event-driven programs like network servers or Tk applications should not block for long periods of time while executing any one operation, because this prevents other events from being serviced. For example, if a Tk application attempts to read from its standard input at a time when no input is available, the application blocks until input appears. During this time the process is suspended by the operating system so it cannot service GUI events. This means, for example, that the application cannot respond to mouse actions or redraw itself. Such behavior is likely to be annoying to users, since they expect to be able to interact with the application at any time.

Channel handlers provide an event-driven mechanism for reading and writing channels that have long I/O delays. The function `Tcl_CreateChannelHandler` creates a new file handler:

```
void Tcl_CreateChannelHandler(Tcl_Channel channel,
        int mask, Tcl_FileProc *callback,
        ClientData clientData)
```

The `channel` argument gives the handle for an existing Tcl channel and may refer to many different types of channel (e.g., network sockets, serial lines, pipes, etc.), and `mask` indicates when `callback` should be invoked. It is an ORed combination of the following bits:

- `TCL_READABLE`—Tcl should invoke `callback` whenever data is waiting to be read on `channel`.

- `TCL_WRITABLE`—Tcl should invoke `callback` whenever `channel` is capable of accepting more output data.
- `TCL_EXCEPTION`—Tcl should invoke `callback` whenever an exceptional condition is present for `channel`.

The `channel` argument must match the following prototype:

```
typedef void Tcl_FileProc(ClientData clientData, int mask);
```

The `clientData` argument is the same as the `clientData` argument to `Tcl_CreateChannelHandler`, and `mask` contains a combination of the bits `TCL_READABLE`, `TCL_WRITABLE`, and `TCL_EXCEPTION` to indicate the state of the channel at the time of the callback. You can place as many different callbacks on a channel at once as you wish as long as they all have distinct `callback` and `clientData` values.
To delete a file handler, call `Tcl_DeleteChannelHandler` with the same `channel`, `callback`, and `clientData` arguments that were used to create the handler:

```
void Tcl_DeleteChannelHandler(Tcl_Channel channel,
      Tcl_FileProc *callback, ClientData clientData)
```

Note

You can temporarily disable a particular channel callback by calling `Tcl_ CreateChannelHandler` with a mask of `0`. You then can call `Tcl_CreateChannelHandler` again to reset the mask when you want to re-enable the handler. This approach is more efficient than calling `Tcl_DeleteChannelHandler` to delete the handler.

With channel handlers you can do event-driven file I/O. Instead of opening a channel, reading it from start to finish, and then closing the channel, you open the channel, create a channel handler for it, and then return. When the channel is readable, the callback is invoked. It issues exactly one read request for the channel, processes the data returned by the read, and then returns. When the channel becomes readable again (perhaps immediately), the callback is invoked again. Eventually, when all of the data from the channel has been read, the channel becomes readable and the read call returns an end-of-file condition. At this point the channel can be closed and the channel handler deleted. With this approach, your application is still able to respond to other incoming events even if there are long delays in

reading the channel.
For example, `wish` uses a file handler to read commands from its standard input when it is connected to a real console. The main program for `wish` creates a channel handler for standard input (file descriptor 0) with the following statement:

```
...
Tcl_Channel inChan = Tcl_GetStdChannel(TCL_STDIN);
Tcl_CreateChannelHandler(inChan, TCL_READABLE,
        StdinProc, (ClientData) inChan);
Tcl_DStringInit(&command);
Tcl_DStringInit(&line);
...
```

In addition to registering `StdinProc` as the callback for standard input, this code initializes a dynamic string that is used to buffer lines of input until a complete Tcl command is ready for evaluation, and another dynamic string that is used to read lines of text from the channel. The main program enters the event loop as described in Section 43.5. When data becomes available on standard input, `StdinProc` is invoked. Its code[1] is as follows:

1. The real implementation of `StdinProc` is considerably more complex because of the use of thread-specific data to enable multithreaded operation of Tk.

```
void StdinProc(ClientData clientData, int mask) {
    Tcl_Channel chan = (Tcl_Channel) clientData;
    Tcl_DString line, command;
    char *cmdStr;
    int code, count;
```

```
        count = Tcl_Gets(chan, &line);
        if (count < 0) {
            ... handle errors and end of file ...
        }
        Tcl_DStringAppend(&command,
                Tcl_DStringValue(&line), -1);
        cmdStr = Tcl_DStringAppend(&command, "\n", -1);
        Tcl_DStringFree(&line);
        if (Tcl_CommandComplete(cmdStr)) {
            Tcl_CreateChannelHandler(chan, 0,
                    StdinProc, chan);
            code = Tcl_Eval(interp, cmdStr);
            Tcl_CreateChannelHandler(chan, TCL_READABLE,
                    StdinProc, chan);
            Tcl_DStringFree(&command);
            ...
        }
        ...
    }
```

After reading from standard input and checking for errors and end of file, `StdinProc` adds the new data to the command buffer's current contents. Then it checks to see if the buffer contains a complete Tcl command (it won't, for example, if a line such as `foreach i $x {` has been entered but the body of the `foreach` loop hasn't yet been typed). If the command is complete, `StdinProc` disables the channel handler, evaluates the command, reestablishes the handler, clears the dynamic string buffer, and is ready for the next command.

Note

It is usually best to use non-blocking I/O with file handlers, just to be absolutely sure that I/O operations don't block. To request non-blocking I/O, pass the correct flag bit to `Tcl_FSOpenFileChannel`, and set the `async` flag to `Tcl_OpenTcpClient`, or use `Tcl_SetChannelOption` to turn off blocking. If you use channel handlers for writing to channels with long output delays, such as pipes and network sockets, it is essential that you use non-blocking I/O; otherwise, if you supply too much data in a `Tcl_Write` call, the output buffers fill and the operating system puts the process to sleep.

Note

For ordinary disk files, you should not use the event-driven approach described in this section, because reading and writing these files rarely incurs noticeable delays, and operating systems maintain a fiction that they never do. Channel handlers are useful primarily for channels such as terminals, pipes, and network connections, which can block for indefinite periods of time. Unfortunately, with network-based files the fiction of immediate action wears rather thin, but operating systems still maintain the fiction and only allow you to do blocking waits for them. Sometimes abstractions are inevitably incomplete.

43.3 Timer Events

Timer events trigger callbacks after particular time intervals. For example, entry widgets use timer events to display blinking insertion cursors, and the `http` package uses timer events to time out very slow connections. In the case of entry widgets, when the entry gains the input focus, it displays the insertion cursor and creates a timer callback that triggers in a few tenths of a second. The timer callback erases the insertion cursor and reschedules itself for a few tenths of a second later. The next time the callback is invoked, it turns the insertion cursor on again. This process repeats indefinitely so that the cursor blinks on and off. When the widget loses the input focus, it cancels the timer callback and erases the insertion cursor.
The function `Tcl_CreateTimerHandler` creates a timer callback:

```
Tcl_TimerToken Tcl_CreateTimerHandler(int milliseconds,
            Tcl_TimerProc *callback, ClientData clientData);
```

The *`milliseconds`* argument specifies how many milliseconds should elapse before the callback is invoked. `Tcl_CreateTimerHandler` returns immediately, and its return value is a token that can be used to cancel the callback. After the given interval has elapsed, Tcl invokes *`callback`*, which must match the following prototype:

```
void Tcl_TimerProc(ClientData clientData);
```

Its argument is the *`clientData`* passed to `Tcl_CreateTimerHandler.` *`callback`* is called only once, after which Tcl deletes the callback automatically. If you want *`callback`* to be called over and over at regular intervals, *`callback`* should reschedule itself by calling `Tcl_CreateTimerHandler` each time it is invoked.

Note

There is no guarantee that *`callback`* will be invoked at exactly the specified time. If the application is busy processing other events at the specified time, *`callback`* won't be invoked until the next time the application invokes the event dispatcher, as described in Section 43.5.

The function `Tcl_DeleteTimerHandler` cancels a timer callback:

```
void Tcl_DeleteTimerHandler(Tcl_TimerToken token);
```

It takes a single argument, which is a token returned by a previous call to `Tcl_CreateTimerHandler`, and deletes the callback so that it is never invoked. It is safe to invoke `Tcl_DeleteTimerHandler` even if the callback has already been invoked; in this case the function has no effect.

43.4 Idle Callbacks

The function `Tcl_DoWhenIdle` creates an *idle callback*:

```
void Tcl_DoWhenIdle(Tcl_IdleProc *callback,
            ClientData clientData);
```

This arranges for *`callback`* to be invoked the next time the application becomes idle. The application is idle when Tcl's main event-processing function, `Tcl_DoOneEvent`, is called and no channel events or timer events are ready. Normally when this occurs, `Tcl_DoOneEvent` suspends the process until an event occurs. However, if idle callbacks exist, all of them are invoked. Idle callbacks are also invoked when the `update` Tcl command is invoked. The *`callback`* for an idle callback must match the following prototype:

```
typedef void Tcl_IdleProc(ClientData clientData);
```

It returns no result and takes a single argument, which is the `clientData` argument passed to `Tcl_DoWhenIdle`.
`Tcl_CancelIdleCall` deletes an idle callback so that it won't be invoked after all:

```
void Tcl_CancelIdleCall(Tcl_IdleProc *callback,
        ClientData clientData);
```

`Tcl_CancelIdleCall` deletes all of the idle callbacks that match `callback` and `clientData` (there can be more than one). If there are no matching idle callbacks, the function has no effect.

Idle callbacks are used heavily to implement the delayed operations described in Section 29.3. The most common uses of idle callbacks in Tk are for widget redisplay and geometry recalculation. This is because it is generally a bad idea to redisplay a widget immediately when its state is modified, since this can result in multiple redisplays when related modifications happen together. For example, suppose the following Tcl commands are invoked to change a label widget `.l`:

```
.l configure -background purple -textvariable msg
set msg "Hello, world!"
.l configure -bd 3m
```

Each of these commands modifies the widget in a way that requires it to be redisplayed, but it would be a bad idea for each command to redraw the widget. This would result in three redisplays, which are unnecessary and can cause the widget to flash as it steps through a series of changes. It is much better to wait until all of the commands have been executed and then redisplay the widget once. Idle callbacks provide a way of knowing when all available events have been fully processed.

43.5 Invoking the Event Dispatcher

The preceding sections described the first two parts of event management: creating event handlers and writing callback functions. The final part of event management is invoking the Tcl event dispatcher, which waits for events to occur and invokes the appropriate callbacks. If you don't invoke the dispatcher, no events are processed and no callbacks are invoked.

Tcl provides one function for event dispatching, `Tcl_DoOneEvent`, and Tk provides another, `Tk_MainLoop`. Tk applications normally use `Tk_MainLoop`, and non-Tk applications always use `Tcl_DoOneEvent`, usually within some kind of loop.

Note

Tcl applications can install a main-loop function into Tcl with `Tcl_SetMainLoop`. `Tcl_Main` calls the installed main-loop function after the initial startup script (i.e., the script file specified on the command line) has been executed. Tk installs `Tk_MainLoop` as a main-loop function by default; if you want to use a custom main-loop function but also use Tk, make sure you install your function only after the Tk package is loaded.

`Tk_MainLoop` takes no arguments and returns no result; it is typically invoked once, in the main program after initialization. `Tk_MainLoop` calls the Tcl event dispatcher repeatedly to process events. When all available events have been processed, it suspends the thread until more events occur, and it repeats this over and over. It returns only when every Tk window created by the thread has been destroyed (for example, after the `destroy .` command has been executed). A typical main program for a Tk application creates a Tcl interpreter, creates the main Tk window and application, performs some other application-specific initialization (such as evaluating a Tcl script to create the application's interface), and then calls `Tk_MainLoop`. When `Tk_MainLoop` returns, the main program exits. Thus Tk provides top-level control over the application's execution, and all of the application's useful work is carried out by event handlers invoked via `Tk_MainLoop`.
The other function for event dispatching is `Tcl_DoOneEvent`, which provides a lower-level interface to the event dispatcher:

int Tcl_DoOneEvent(int *flags*)

The *`flags`* argument is normally `0` (or, equivalently, `TCL_ALL_EVENTS`). In this case `Tcl_DoOneEvent` processes a single event and then returns `1`. If no events are pending, `Tcl_DoOneEvent` suspends the thread until an event arrives, processes that event, and then returns `1`.
For example, `Tk_MainLoop` is implemented using `Tcl_DoOneEvent`:

```
void Tk_MainLoop(void) {
    while (/* Number of main windows > 0 */) {
        Tcl_DoOneEvent(0);
    }
}
```

As you can see, `Tk_MainLoop` just calls `Tk_DoOneEvent` over and over until all the main windows have been deleted.
`Tk_DoOneEvent` is also used by commands such as `vwait` and `tkwait` that want to process events while waiting for something to happen. For example, the

`vwait` command processes events until a given variable is updated, then it returns. Here is the C code that implements this command:

```
int done;
...
Tcl_TraceVar(interp, nameString,
    TCL_GLOBAL_ONLY|TCL_TRACE_WRITES|TCL_TRACE_UNSETS,
    VwaitVarProc, &done);
done = 0;
while (!done) {
    Tcl_DoOneEvent(0);
}
Tcl_UntraceVar(interp, nameString,
    TCL_GLOBAL_ONLY|TCL_TRACE_WRITES|TCL_TRACE_UNSETS,
    VwaitVarProc, &done);
}
...
```

The variable `nameString` identifies the variable whose update is awaited. The code creates a trace callback that is invoked when the variable is updated, then invokes `Tcl_DoOneEvent` over and over until the `done` flag is set to indicate that the variable has been set or deleted. The callback for the variable trace is as follows:

```
char *VwaitVarProc(ClientData clientData,
    Tcl_Interp *interp, const char *name1,
    const char *name2, int flags)
{
    int *donePtr = clientData;|

    *donePtr = 1;
    return NULL;
}
```

The `clientData` argument is a pointer to the flag variable. `VwaitVarProc` is only ever called in the situation that it has been set to detect, so all it has to do is set the flag variable to `1` when it is called.
The `flags` argument to `Tcl_DoOneEvent` can be used to restrict the kinds of events it considers. If it contains any of the bits `TCL_FILE_EVENTS`[2], `TCL_TIMER_EVENTS`, or `TCL_IDLE_EVENTS`, only the events indicated by the specified bits are considered. Furthermore, if the `flags` argument includes the bit `TCL_DONT_WAIT`, or if neither file events nor timer events are selected, `Tcl_DoOneEvent` doesn't suspend the thread if no event is ready to be processed. Instead, it returns immediately with a `0` result to indicate that it had nothing to do. For example, the `update idletasks` command is implemented with the following code:

2. Tk's X events are implemented as `TCL_FILE_EVENTS` in the lowest levels of the event dispatcher by directly observing the communication channel that the X library uses to communicate with the X server. This is fairly tricky code, but it allows the user of Tk, even as a C library, to ignore the details.

```
while (Tcl_DoOneEvent(TCL_IDLE_EVENTS) != 0) {
    /* empty loop body */
}
```

By comparison, the core of the normal `update` command is this code:

```
while (Tcl_DoOneEvent(TCL_ALL_EVENTS|TCL_DONT_WAIT)) {
    /* empty loop body */
}
```

44. File System Interaction

Tcl provides a platform-independent C API for working with files and directories. Using these functions instead of platform-native system calls makes your code more portable. The Tcl API also automatically handles any translation required between the Tcl-native UTF representation of character strings and the system encoding; all input and output string parameters for these functions are based on Tcl-native UTF-encoded strings. Furthermore, the Tcl API automatically supports Tcl's virtual file systems. If the paths you provide as parameters to these functions represent locations in a virtual file system that your application has mounted (for example, a ZIP archive or an FTP site), the functions can access the files as though they were typical on-disk file systems.

Note

Most of these functions have an equivalent Tcl command at the scripting level. In general, consider using a Tcl script if possible when interacting with the file system, as it is faster to develop and easier to modify in the future, should it prove necessary. These C-level functions are most appropriate for performing occasional file system operations when doing significant development in C.

44.1 Tcl File System Functions

Table 44.1 lists the functions provided for file system access and their equivalent scripting-level commands. You can find more details about them in Tcl's `FileSystem.3` reference documentation. Most of these functions return `0` on success or `-1` on failure and set `errno` to describe the reason for failure. Additionally, Tcl's file system API is based almost completely on Tcl objects. Some functions may cache internal representations and other path-related strings, so objects that you pass to these functions must have a reference count greater than 0, for example:

```
Tcl_Obj *fileName = Tcl_NewStringObj("/tmp/foobar", -1);
Tcl_IncrRefCount(fileName);
ret = Tcl_FSDeleteFile(fileName);
Tcl_DecrRefCount(fileName);
```

Table 44.1 Tcl C Functions and Equivalent Tcl Commands for File System Interaction

C function	Equivalent Tcl command

`Tcl_FSAccess`	`file readable`
	`file writable`

`Tcl_FSChdir`	`cd`
`Tcl_FSCopyDirectory` `Tcl_FSCopyFile`	`file copy`
`Tcl_FSCreateDirectory`	`file mkdir`
`Tcl_FSEvalFile`	`source`

`Tcl_FSFileAttrsGet`
`Tcl_FSFileAttrsSet`
`Tcl_FSFileAttrStrings`
	`file attributes`

`Tcl_FSFileSystemInfo`	`file system`
`Tcl_FSGetCwd`	`pwd`
`Tcl_FSGetNormalizedPath`	`file normalize`
`Tcl_FSGetPathType`	`file pathtype`

`Tcl_FSJoinPath` `Tcl_FSJoinToPath`	`file join`

`Tcl_FSLink`	`file link`
	`file readlink`

`Tcl_FSListVolumes`	`file volumes`
`Tcl_FSLoadFile`	`load`
`Tcl_FSLstat`	`file lstat`
`Tcl_FSMatchInDirectory`	`glob`
`Tcl_FSOpenFileChannel`	`open`
`Tcl_FSPathSeparator`	`file separator`
`Tcl_FSRemoveDirectory` `Tcl_FSDeleteFile`	`file delete`
`Tcl_FSRenameFile`	`file rename`
`Tcl_FSSplitPath`	`file split`
`Tcl_FSStat`	`file stat`
`Tcl_FSUtime`	`file mtime`
`Tcl_TranslateFileName`	`file nativename`

If a function returns an object, you should assume that the reference count is 0. As with any other newly created Tcl object, if you use the object elsewhere, you might need to call `Tcl_IncrRefCount` on the object to make sure that it isn't freed accidentally in the wrong place. See Chapter 32 for more information on managing the reference counts of Tcl objects.

44.2 Virtual File Systems

Another interesting aspect of the Tcl file system interaction implementation is that it is built on top of a "virtual file system" that is accessible via Tcl's C API. In other words, it is possible to write new "file systems" that Tcl then accesses as if they were regular files. Some examples of this are archive files, such as ZIP and tar, and remote files—WebDAV, FTP, and HTTP. When you "mount" a ZIP file with the appropriate extension, you can `cd` inside it, `open` files, use `glob`, and even `source` Tcl files inside it, all as if it were just a part of the disk in which Tcl resides. Chapter 11 discusses how you can use the `tclvfs` extension to mount and access such virtual file systems in your applications.
Detailed coverage of the process of implementing a new virtual file system is beyond the scope of this book and may be found in Tcl's excellent reference documentation—specifically, the `FileSystem.3` reference page. The

basic process is similar to that covered in Chapter 42 for implementing custom channel types: you create a structure and set its fields to functions that carry out the underlying actions of the specific operations described in this chapter.

45. Operating System Utilities

Tcl provides a variety of functions to deal with underlying operating system services. This layer is platform-independent, except where it makes sense to give the user access to lower-level options that may not be present on all platforms.

45.1 Functions Presented in This Chapter

This chapter discusses the following functions for accessing operating system services:

- `Tcl_Channel Tcl_OpenCommandChannel(Tcl_Interp *`*`interp`*`,`
  `int `*`argc`*`, CONST char *`*`argv`*`, int `*`flags`*`)`

Opens a command channel using *`argv`* as the command and its arguments.

- `Tcl_DetachPids(int `*`numPids`*`, int *`*`pidPtr`*`)`

Gives Tcl responsibility for *`numPids`* child processes that are passed in via the *`pidPtr`* array.

- `Tcl_ReapDetachedProcs()`

Invokes the `waitpid` system call on each background process so that its state can be cleaned up if it has exited. If the process hasn't exited yet, `Tcl_ReapDetachedProcs` doesn't wait for it to exit.

- `Tcl_Pid Tcl_WaitPid(int `*`pid`*`, int *`*`statPtr`*`, int `*`options`*`)`

Wrapper for the `waitpid` system call.

- `Tcl_AsyncHandler Tcl_AsyncCreate(Tcl_AsyncProc *`*`func`*`,`
  `ClientData `*`clientData`*`)`

Creates an asynchronous handler, which is able to interrupt Tcl execution, and returns a token for it.

- `Tcl_AsyncDelete(Tcl_AsyncHandler `*`async`*`)`

Deletes an asynchronous handler, given a token.

- `Tcl_AsyncMark(Tcl_AsyncHandler `*`async`*`)`

Marks the asynchronous handler indicated by *`async`* as ready to run.

- `int Tcl_AsyncInvoke(Tcl_Interp *`*`interp`*`, int `*`code`*`)`

Calls all handlers that are ready.

- `int Tcl_AsyncReady()`

Checks to see if any handlers are ready to be run.

- `CONST char *Tcl_SignalId(int `*`sig`*`)`

Returns a machine-readable string describing the signal, such as `SIGPIPE`.

- `CONST char *Tcl_SignalMsg(int *sig*)`

Returns a human-readable string describing the signal, such as `bus error`.

- `Tcl_Exit(int *status*)`

Exits Tcl and the process.

- `Tcl_Finalize()`

Similar to `Tcl_Exit`, but only ends the use of Tcl; it does not terminate the entire process.

- `Tcl_CreateExitHandler(Tcl_ExitProc *proc*,`

  `ClientData *clientData*)`

Creates a handler to be called when `Tcl_Exit` or `Tcl_Finalize` is called.

- `Tcl_DeleteExitHandler(Tcl_ExitProc *proc*,`

  `ClientData *clientData*)`

Deletes an exit handler created by `Tcl_CreateExitHandler`.

- `CONST char* Tcl_GetHostName()`

Returns a pointer to a string containing the current host name.

- `Tcl_GetTime(Tcl_Time **timePtr*)`

Fills in the previously allocated *`timePtr`* structure, which contains the number of seconds since the epoch and the number of microseconds since the last second.

- `int Tcl_PutEnv(const char **assignment*)`

Given an *`assignment`* string in the form *`NAME`*`=`*`value`*, sets an environment variable.

45.2 Processes

The `Tcl_OpenCommandChannel` function spawns a process external to Tcl—the equivalent of evaluating an `exec` or `open |` command. This is another case where it typically makes sense to write Tcl code instead of C code if possible, because the functionality is virtually identical, and writing Tcl code is much faster. Its syntax is

```
Tcl_Channel Tcl_OpenCommandChannel(
    Tcl_Interp *interp, int argc,
    CONST char **argv, int flags);
```

`argc` is the number of arguments passed to the command or commands, via the array of strings *`argv`*. *`argv`*`[`*`argc`*`]` must be `NULL`, so be sure to make *`argv`* one element larger than *`argc`*. Keep in mind that it is possible to launch a pipelined sequence of commands such as `ls`, `|`, `grep`, and `foo`, and that standard `exec` arguments (see Chapter 12) like `>`, `<`, `&`, and so on are also valid

elements of `argv`. The channel returned is affected by an ORed combination of the following flags:

- `TCL_STDIN`

The newly created channel is attached to the standard input of the first process in the sequence launched by `Tcl_OpenCommandChannel`. In other words, data written to the channel is sent as input to the process. If this flag is not set, the first process uses the current (calling) process's standard input.

- `TCL_STDOUT`

The newly created channel is attached to the standard output of the last process launched by `Tcl_OpenCommandChannel`. Reading from the channel returns data from the last process created. If this flag is not set, the last process's standard output goes to the standard output of the (current) calling process.

- `TCL_STDERR`

Data written to the standard error channel by any process spawned by `Tcl_OpenCommandChannel` is directed to the channel for reading. When the channel is closed, if there is data in the standard error channel, an error is raised. If this flag is not set, the standard error data goes to the standard error of the (current) calling process.

- `TCL_ENFORCE_MODE`

This flag, when it is set, means that standard `exec` arguments such as `<` and `>` cannot override the `TCL_STDIN`, `TCL_STDOUT`, and `TCL_STDERR` flags; an error is raised if redirections are attempted.

Note

The newly created channel is not registered with the supplied interpreter. See Chapter 42 for information on registering channels.

In the following example we create a `subtcl` command, which pipes a script into a `tclsh` that is launched as a subprocess and then returns the output:

```
static int
SubTclCmd(ClientData clientData, Tcl_Interp *interp,
        int objc, Tcl_Obj *CONST objv[]) {
    Tcl_Obj *out;
    CONST char **argv;
    Tcl_Channel chan;
    int argc = 1;
    static char *argv0 = NULL;

    if (objc != 2) {
        Tcl_WrongNumArgs(interp, 1, objv, "script");
        return TCL_ERROR;
    }
    if (argv0 == NULL) {
        Tcl_Eval(interp, "info nameofexecutable");
        argv0 = strdup(Tcl_GetStringResult(interp));
    }



    out = Tcl_NewObj();
    argv = (CONST char **)ckalloc(sizeof(char *) * 2);
    argv[0] = argv0;
    argv[1] = NULL;
    chan = Tcl_OpenCommandChannel(
        interp, argc, argv,
        TCL_STDOUT|TCL_STDIN|TCL_STDERR);
    ckfree((char *)argv);

    if (chan == NULL) {
        return TCL_ERROR;
    }
    if (Tcl_WriteObj(chan, objv[1]) < 0) {
        return TCL_ERROR;
    }
    if (Tcl_Flush(chan) != TCL_OK) {
        return TCL_ERROR;
    }
    if (Tcl_ReadChars(chan, out, -1, 0) < 0) {
        return TCL_ERROR;
    }
    if (Tcl_Close(interp, chan) != TCL_OK) {
        return TCL_ERROR;
    }
    Tcl_SetObjResult(interp, out);
    return TCL_OK;
}
```

We construct the command in `argv`, then pass it to `Tcl_OpenCommandChannel`. The subprocess is able to read the input because we set the `TCL_STDIN` flag and then send `objv[1]` to the channel with `Tcl_WriteObj`, followed by a flush operation to make sure that the data has arrived. Thanks to the `TCL_STDOUT` flag, the subprocess returns its results to `chan`, where we subsequently read them and return them as the result of the command. The `subtcl` command is called from Tcl like so:

```
set result [subtcl {
    puts hello
    exit
}]
```

The `subtcl` command returns any output from the `tclsh` process it spawns. One improvement that could be made to this example would be to involve the event loop so that the calling process would not block while waiting for the subprocess to finish.

45.3 Reaping Child Processes

`Tcl_OpenCommandChannel` is sufficient to manage the lifetime of external processes. However, if you wish to launch external processes yourself, such as by using the `fork` and `exec` system calls, you can still manage them from Tcl.

If an application creates a subprocess and abandons it (i.e., the parent never invokes a system call to wait for the child to exit), the child executes in background, and when it exits it becomes a *zombie*. It remains a zombie until its parent officially waits for it or until the parent exits. Zombie processes occupy space in the system's process table, so if you create enough of them, you can overflow the process table and make it impossible for anyone to create more processes. To keep this from happening, you must invoke a system call such as `wait` or `waitpid`, which returns the exit status of the zombie process. Once the status has been returned, the zombie relinquishes its slot in the process table. This procedure is often referred to as *reaping* the child processes.

The `Tcl_DetachPids` function notifies Tcl to take responsibility for reaping one or more child processes:

```
Tcl_DetachPids(int numPids, int *pidPtr);
```

The `pidPtr` argument is an array of process identifiers, and `numPids` is the number of identifiers in the array. Each of these processes now becomes the property of Tcl, and the caller should not refer to them again.
To reap these child processes, you call `Tcl_ReapDetachedProcs`, which takes care of the low-level work of cleaning up after any processes that have exited. If some of the detached processes are still executing, `Tcl_ReapDetachedProcs` doesn't actually wait for them to exit; it cleans up only the processes that have already exited and returns.
Should it prove useful to wait on one particular child process, the `Tcl_WaitPid` function, a wrapper for the `waitpid` system call, is the right tool for the job:

```
Tcl_Pid Tcl_WaitPid(int pid, int *statPtr, int options);
```

The `pid` argument is the process identifier, and `statPtr` is a pointer to an integer that the function sets to either `0` or `ECHILD` to indicate the status of the child process. The `options` permitted depend on the underlying operating system, but `WNOHANG` should indicate that the function should return immediately, rather than block, if the child has not exited.

45.4 Asynchronous Events

Tcl does not provide commands for handling ANSI or POSIX signals (though you can use an extension such as TclX or Expect if you need this functionality), but it does provide a generic framework for dealing with asynchronous events like signals. The central concept in this infrastructure is that when a signal (or some other event) arrives, it should not interrupt Tcl while Tcl is busy. For instance, if a signal arrives and you attempt to perform an `eval` while Tcl is already performing that command, instability could result. Instead, Tcl is notified that an event of interest has occurred via a flag, and when Tcl is able to, it processes the event.
The following example demonstrates an extremely simple signal handler:

```
#include <tcl.h>
#include <signal.h>

static Tcl_AsyncHandler SignalHandler = NULL;
static int sigs = 0;    /* Number of signals received. */

int Signals_Init(Tcl_Interp *interp) {

#ifdef USE_TCL_STUBS
    if (Tcl_InitStubs(interp, "8", 0) == NULL) {
        return TCL_ERROR;
    }
#endif

    Tcl_SetVar2Ex(interp, "::counter", NULL,
                  Tcl_NewIntObj(0), 0);

    SignalHandler = Tcl_AsyncCreate(HandleSignals,
                                    (ClientData)interp);
    signal(SIGHUP, lowlevelhandler);
    signal(SIGINT, lowlevelhandler);

    if (Tcl_PkgProvide(interp, "signals", "0.1") != TCL_OK) {
        return TCL_ERROR;
    }
    return TCL_OK;
}
```

This code uses the `Tcl_AsyncCreate` function to register a `Tcl_AsyncHandler` function named `HandleSignals`. The return value is a unique token to identify the handler. As is discussed below, an interpreter is not always available to asynchronous handlers, so we pass the current interpreter to it as client data. This carries some risk, in that the interpreter might be deleted before the handler is called, but for the sake of our example, we assume this is not the case. The next two lines initialize signal handlers for `SIGHUP` (hangup) and `SIGINT` (interrupt), which use this function:

```
void lowlevelhandler(int signum) {
     sigs++;
     Tcl_AsyncMark(SignalHandler);
}
```

The `lowlevelhandler` function simply increments the tally of received signals in a global variable and then invokes `Tcl_AsyncMark` to tell Tcl that `SignalHandler`'s callback is ready for action. When Tcl reaches a point in its execution where it is able to process the asynchronous event, it calls the `HandleSignals` function:

```
static int
HandleSignals(ClientData clientData,
              Tcl_Interp *interp, int code) {
    int i = 0;
    int counterVal = 0;
    Tcl_Obj *counterObj;
    if (interp == NULL) {
        interp = (Tcl_Interp *)clientData;
        if (interp == NULL) {
            /* It's deleted, can't do anything useful. */
            return TCL_ERROR;
        }
    }

    counterObj = Tcl_GetVar2Ex(interp, "::counter", NULL, 0);
    Tcl_GetIntFromObj(interp, counterObj, &counterVal);
    while (sigs) {
        Tcl_SetVar2Ex(interp, "::counter", NULL,
                      Tcl_NewIntObj(counterVal+1), 0);
        i++;
        sigs--;
    }
    return code;
}
```

The first argument to the signal handler corresponds to the `clientData` argument of the `Tcl_AsyncCreate` function and, as in other Tcl subsystems, serves as a means to pass data to the function called by the asynchronous handler. If the handler is invoked just after an interpreter completes execution of a command, the `interp` argument identifies the interpreter in which the command was evaluated, and `code` is the completion code returned by that command. (The command's result is present in the interpreter's result.) When the handler returns, whatever it leaves in the interpreter's result is returned as the result of the command, and the integer return value of the handler is the new completion code for the command. On the other hand, if Tcl is in the event loop or in other states where no interpreter is active, `interp` is `NULL` and `code` is `0`, and the return value from the handler is ignored.

Note

It is almost always a bad idea for an asynchronous event handler to modify the interpreter's result or return a code different from its `code` argument. This sort of behavior can disrupt the execution of scripts in subtle ways and result in bugs that are extremely difficult to track down. If an asynchronous event handler needs to evaluate Tcl scripts, it should first save the interpreter's state by calling `Tcl_SaveInterpState`, passing in the `code` argument. When the asynchronous handler is finished, it should restore the interpreter's state by calling `Tcl_RestoreInterpState` and then returning the `code` argument.

In this simple example, we first attempt to obtain a valid interpreter, returning an error condition if we don't have one. Then we increment a counter variable for each signal that has arrived.

Note

Even in cases where the handler receives an active interpreter as an argument, there are no guarantees as to which interpreter it is, if several are active, or in what sort of state it is. Thus, it is recommended that you do not attempt to do too much even in this handler. One possible solution would be to make asynchronous events into an event source for Tcl's event loop. See Chapter 42 for more information.

45.5 Signal Names

On the topic of signals, Tcl also provides two functions that let you transform a signal number into something more readable. Both `Tcl_SignalId` and `Tcl_SignalMsg` take a POSIX signal number as argument, and each returns a string describing the signal. `Tcl_SignalId` returns the official POSIX name for the signal as defined in `signal.h`, and `Tcl_SignalMsg` returns a human-readable message describing the signal. For example,

```
Tcl_SignalId(SIGALRM)
```

returns the string `SIGALRM`, and

```
Tcl_SignalMsg(SIGALRM)
```

returns `alarm` clock.

45.6 Exiting and Cleanup

The `Tcl_Exit` function call is provided to cleanly exit a process. It is equivalent to the `exit` command in Tcl. Never call the C `exit` function directly; Tcl provides a callback mechanism for exit handlers via `Tcl_CreateExitHandler`. `Tcl_Exit` ensures that the proper cleanup is performed, whereas simply calling `exit` may skip that step. Like other callbacks, this one takes a function as an argument:

```
static void
OnExit(ClientData clientData) {
    fprintf(stderr, "We are down for the count\n");
}

...

Tcl_CreateExitHandler(OnExit, NULL);
```

It is possible to provide a `clientData` value that is passed to the exit handler. This could be used to cleanly shut down any communications with other systems, networked resources such as a database server, connected clients, and so forth. You can delete an existing exit handler with the `Tcl_DeleteExitHandler` function.
Should you wish only to clean up Tcl in your program and continue running, instead of shutting down the whole process, the `Tcl_Finalize` function carries out the necessary work. This call should also be used prior to unloading, if Tcl was loaded into your application as part of a shared object (`.so` or `.dll`) that at some point may be unloaded. Most applications load modules but do not unload them, so this is not a very common occurrence.

Note

If you are writing a multithreaded Tcl application, there are analogous functions for exiting a thread, terminating the use of Tcl within a specific thread, and registering thread exit handlers. See Chapter 46 for more information.

45.7 Miscellaneous

The `Tcl_GetHostName` function returns a string that contains the host name for the machine on which Tcl is running. The storage allocated belongs to Tcl, and you should not attempt to free it.
The `Tcl_PutEnv` function is meant to be a replacement for the `putenv` system call. It takes a string argument of the form *`VARNAME=value`*. Remember that it is also possible to work with environmental variables via Tcl's global `env` array, using `Tcl_SetVar` and the like.
To obtain information about the current time, Tcl provides the `Tcl_GetTime` function, which retrieves a structure consisting of two long integers: the current time in seconds since January 1, 1970, 00:00 UTC epoch, and the number of microseconds that have elapsed since the start of the current second; for example:

```
Tcl_Time timePtr;
...
Tcl_GetTime(&timePtr);
printf("seconds %ld\n", timePtr.sec);
printf("microseconds %ld\n", timePtr.usec);
```

46. Threads

This chapter covers the use of Tcl's thread API from C and the use of Tcl in threaded programs. Complex programming with threads is tricky and can lead to difficult-to-find errors if not treated with caution. We discuss Tcl's API but do not cover the nuances of programming with threads. For a thorough discussion of the topic, refer to books such as *Programming with POSIX Threads* by David Butenhof (ISBN 0-201-63392-2).

46.1 Functions Presented in This Chapter

This chapter discusses the following functions related to thread management in Tcl:

- `int Tcl_CreateThread(Tcl_ThreadId *idPtr,`
  `Tcl_ThreadCreateProc threadProc,`
  `ClientData clientData, int stackSize, int flags)`

Creates a new thread whose new ID is stored in `idPtr` and starts running in the function `threadProc`. See the text and the reference documentation for information on the `stackSize` and `flags` arguments.

- `Tcl_ExitThread(int status)`

Terminates the current thread and invokes per-thread exit handlers.

- `Tcl_FinalizeThread()`

Cleans up per-thread Tcl state and calls thread exit handlers without actually terminating the thread.

- `Tcl_CreateThreadExitHandler(Tcl_ExitProc proc,`
  `ClientData clientData)`

Registers a per-thread exit handler `proc` that is called with `clientData`.

- `Tcl_DeleteThreadExitHandler(Tcl_ExitProc proc,`
  `ClientData clientData)`

Deletes a per-thread exit handler.

- `int Tcl_JoinThread(Tcl_ThreadId id, int result)`

Waits on the exit of a thread that has been marked joinable via the `TCL_THREAD_JOINABLE` flag to `Tcl_CreateThread`. Attempting to wait on a non-joinable thread returns an error.

- `void *Tcl_GetThreadData(Tcl_ThreadDataKey *keyPtr,`
  `int *size)`

Returns a pointer to a block of thread-private data. Its argument is a key that is shared by all threads and a *size* for the block of storage. The storage is automatically allocated and initialized to all zeros the first time each thread asks for it. The storage is deallocated automatically by `Tcl_FinalizeThread`.

- `TCL_DECLARE_MUTEX(`*`mutexName`*`)`

Macro to declare mutexes in a portable way. Has no effect if threads are not enabled.

- `void Tcl_MutexLock(Tcl_Mutex *`*`mutexPtr`*`)`

Locks a mutex, waiting until it’s unlocked if some other thread holds the lock.

- `void Tcl_MutexUnlock(Tcl_Mutex *`*`mutexPtr`*`)`

Unlocks a mutex.

- `void Tcl_MutexFinalize(Tcl_Mutex *`*`mutexPtr`*`)`

Frees any resources associated with a mutex after it is no longer needed.

- `void Tcl_ConditionNotify(Tcl_Condition *`*`condPtr`*`)`

Unblocks threads waiting on the condition variable.

- `void Tcl_ConditionWait(Tcl_Condition *`*`condPtr`*`,`

  `Tcl_Mutex *`*`mutexPtr`*`, Tcl_Time *`*`timePtr`*`)`

Waits for the condition in *condPtr* to be notified. Returns with *mutexPtr* locked. If *timePtr* is not `NULL`, waits only the specified time for the condition to be notified.

- `void Tcl_ConditionFinalize(Tcl_Condition *`*`condPtr`*`)`

Frees any resources associated with a condition after it is no longer needed.

46.2 Thread Safety

Tcl is thread-safe, meaning that it’s possible to use it within multithreaded programs without problems. However, Tcl interpreters must be confined to and accessed by only one thread, the thread where the interpreter was created. This means that you cannot create an interpreter and share it among several threads.

46.3 Building Threaded Tcl

Currently, most distributions of Tcl are built with threads enabled. You can check by testing for the existence of `::tcl_platform(threaded)` in your interpreter. If you are building Tcl yourself, you must enable a threaded build by running the `./configure` script with `--enable-threads` at build time. See Chapter 47 on building Tcl for more information. If you have a threaded Tcl, keep in mind that you should also build any extensions you use with `--enable-threads` as well.

46.4 Creating Threads

New threads are created via the cross-platform `Tcl_CreateThread` function, which has the following signature:

```
int Tcl_CreateThread(Tcl_ThreadId *idPtr,
                     Tcl_ThreadCreateProc threadProc,
                     ClientData clientData,
                     int stackSize, int flags);
```

The *`idPtr`* is a pointer to an integer where the thread ID of the newly created thread is stored. When the operating system creates the new thread, it starts running in the *`threadProc`* function, with *`clientData`* as its only argument. It is possible to set the stack size for the new thread, but it is recommended to simply use the `TCL_THREAD_STACK_DEFAULT` macro to let the operating system use the standard stack size. Currently, *`flags`* may be either `TCL_THREAD_NOFLAGS`, which specifies default behavior, or `TCL_THREAD_JOINABLE`, which means that the new thread is joinable.
A thread is *joinable* if a second thread can wait for it to exit. Tcl provides the `Tcl_JoinThread` function to accomplish this task. It takes two arguments: the `Tcl_ThreadID` of the thread to join and a pointer to an integer where the exit code for the joined thread is stored. The `Tcl_JoinThread` function blocks until the specified thread exits. Trying to wait on a non-joinable thread or a thread that is already waited upon results in an error. Waiting for a joinable thread that has already exited is possible; the system retains the necessary information until after the call to `Tcl_JoinThread`. This means that not calling `Tcl_JoinThread` for a joinable thread causes a memory leak.

Note

Windows does not currently support joinable threads. Therefore, the `Tcl_JoinThread` function is not supported on that platform, and the `TCL_THREAD_JOINABLE` flag to `Tcl_CreateThread` is ignored.

46.5 Terminating Threads

Tcl provides the `Tcl_ExitThread` function to terminate the current thread. As part of the termination process, Tcl deletes all interpreters and other Tcl resources associated with the thread. You also have the option of registered *exit handlers* for the thread using the `Tcl_CreateExitHandler` function. The exit handlers can do anything that you like on thread termination, such as flushing buffers and freeing global memory.
Tcl also provides a `Tcl_FinalizeThread` function, which deletes the Tcl interpreters and other resources for the thread and invokes the thread exit handlers but does not terminate the thread. Note that `Tcl_ExitThread` calls `Tcl_FinalizeThread` as part of its operation, so there's no need for you to do so explicitly.

46.6 Mutexes

A *mutex* (short for *mut*ual *ex*clusion) is used to limit access to a portion of code to one thread at a time. If you have a section of code that must execute "atomically"—you don't want it to be interrupted or performed by more than one thread at a time—a mutex is most likely what you need to use. Tcl has a platform-independent macro for creating a mutex:

```
TCL_DECLARE_MUTEX(myMutex);
```

The macro also ensures correct mutex handling even when the core is compiled without threads enabled.
Once you have created a mutex, you can use `Tcl_MutexLock` and `Tcl_MutexUnlock` to protect critical sections of your code:

```
Tcl_MutexLock(&myMutex);
/* ... Critical section ... */
Tcl_MutexUnlock(&myMutex);
```

These functions have no effect if Tcl is not compiled with threads. For a threaded application, if one thread locks a mutex, any other thread that attempts to lock it waits (blocks) until the first thread unlocks it. This means that you'll suffer a performance hit if code that is frequently executed is protected by mutexes (although that is certainly preferable to data corruption!), so try to structure your code in such a way that you don't need to share many resources. The `Tcl_MutexFinalize` function frees the resources associated with a mutex when it is no longer needed.

46.7 Condition Variables

Condition variables (also known simply as *conditions*) are another common tool in multithreaded programming. They are used when one thread needs to let another know something about the state of a shared resource.

Condition variables are used in conjunction with a mutex to guard the shared resource. Before a thread waits on the condition variable, it must first lock the mutex guarding the shared resource. Then it calls `Tcl_ConditionWait` to wait for notification from another thread; this also atomically unlocks the mutex. The thread that performs the notification locks the mutex, then calls `Tcl_ConditionNotify` to notify the waiting thread(s). Once it unlocks the mutex, the `Tcl_ConditionWait` function in the waiting thread finally returns, causing it to "wake up." Upon return, `Tcl_ConditionWait` automatically locks the related mutex for the thread that received the notification.

`Tcl_ConditionWait` takes as arguments a pointer to a `Tcl_Condition` structure, a pointer to a mutex, and a time value that tells Tcl how long to wait on the condition before giving up, or to wait forever if its value is `NULL`. The `Tcl_ConditionNotify` function takes a pointer to a `Tcl_Condition` as its only argument. The resources used by a `Tcl_Condition` may be freed by calling `Tcl_ConditionFinalize` when the condition is no longer needed.

A complication arises in that the waiting thread might receive spurious "wake-ups." This is common when you have multiple worker threads waiting on a condition variable for a shared resource like a message queue. When the notification occurs signaling the arrival of a message, the first thread that resumes processes the message. When the other worker threads get the opportunity to respond to the notification, the message has already been consumed, and so there's nothing for them to do. In these cases, the actual call to `Tcl_ConditionWait` is usually done within a loop that checks the state of the shared resource when `Tcl_ConditionWait` returns and calls it again if it needs to wait again.

As an example from Tcl itself, taken from the `Tcl_InitNotifier` function, here is how Tcl goes about setting up the notifier thread that the threaded version of Tcl uses for its event loop:

```
/* Tcl_InitNotifier: */
...

Tcl_MutexLock(&notifierMutex);
if (notifierCount == 0) {
    if (TclpThreadCreate(&notifierThread,
                         NotifierThreadProc, NULL,
                         TCL_THREAD_STACK_DEFAULT,
                         TCL_THREAD_NOFLAGS) != TCL_OK) {
    Tcl_Panic("Tcl_InitNotifier: unable to start notifier thread");
    }
}
notifierCount++;

/*
 * Wait for the notifier pipe to be created.
 */

while (triggerPipe < 0) {
    Tcl_ConditionWait(&notifierCV, &notifierMutex, NULL);
}
Tcl_MutexUnlock(&notifierMutex);
...

/* NotifierThreadProc: */
...
Tcl_MutexLock(&notifierMutex);
triggerPipe = fds[1];

/*
 * Signal any threads that are waiting.
 */

Tcl_ConditionNotify(&notifierCV);
Tcl_MutexUnlock(&notifierMutex);
...
```

In this code, Tcl first locks the `notifierMutex`. If no notifiers exist, it then launches a notifier thread via the private, internal `TclpThreadCreate` function. This new thread proceeds until it hits a `Tcl_MutexLock(&notifierMutex)`, where it waits. The first thread continues on, executing the `Tcl_ConditionWait`, which has the side effect of unlocking the `notifierMutex`, allowing the second thread to continue, which it does until it executes a `Tcl_ConditionNotify` and `Tcl_MutexUnlock`, returning control to the first thread again, which unlocks the mutex for the last time and goes on about its business. As you can see, programming with threads is complicated indeed!

46.8 Miscellaneous

To retrieve the thread ID of the current thread, use this function:

```
Tcl_ThreadId Tcl_GetCurrentThread();
```

To fetch storage space that can be used for per-thread data, use `Tcl_GetThreadData`:

```
void *Tcl_GetThreadData(Tcl_ThreadDataKey *keyPtr,
            int *size)
```

The function returns a pointer to a block of thread-private data. Its argument is a key that is shared by all threads and a *`size`* for the block of storage. The storage is automatically allocated and initialized to all zeros the first time each thread asks for it. It is in some ways similar to `Tcl_Alloc`, but try not to use it if you don't need to. The storage space allocated is reclaimed by `Tcl_FinalizeThread` only when the thread exits.
Tcl's own use of `Tcl_GetThreadData` in the `ThreadSafeLocalTime` function (written by Kevin Kenny) is instructive. The key, `tmKey`, is declared static to the file:

```
static Tcl_ThreadDataKey tmKey;


/* And then, in the ThreadSafeLocalTime function: */

static struct tm *
ThreadSafeLocalTime(timePtr)
    CONST time_t *timePtr; /* Pointer to the number of
                              seconds since the local
                              system's epoch */
{
    /*
     * Get a thread-local buffer to hold the returned
     * time.
     */
    struct tm *tmPtr =
        (struct tm *)Tcl_GetThreadData(
            &tmKey, (int) sizeof(struct tm));
#ifdef HAVE_LOCALTIME_R
    localtime_r(timePtr, tmPtr);
#else
    Tcl_MutexLock(&clockMutex);
    memcpy((VOID *) tmPtr, (VOID *) localtime
           (timePtr), sizeof(struct tm));
    Tcl_MutexUnlock(&clockMutex);
#endif
    return tmPtr;
}
```

The storage space is allocated by `Tcl_GetThreadData` and then filled in by either the `localtime_r` function or the regular `localtime` function while protected by a mutex.

47. Building Tcl and Extensions

This chapter discusses how to compile Tcl, Tcl extensions, and code that embeds Tcl on three platforms: Unix (including Linux and the various BSD systems), Mac OS, and Windows. For most people, compiling Tcl is not really necessary, as there are various ways to get binary distributions, as discussed in Appendix A. However, if you are integrating Tcl into your own program, you need to be familiar with the process. Building extensions to load in Tcl is often the best approach, and this chapter explains how to go about doing so in a portable way.

47.1 Building Tcl and Tk

The source code for Tcl and Tk is maintained on SourceForge, whose main project home page is at `http://tcl.sourceforge.net`. Tcl and Tk are maintained as separate projects on the site. You can download released versions of the source code from the file distribution links, or you can access the CVS repositories via anonymous access as described on the site. Released versions of the sources are also available from `http://www.tcl.tk/software/tcltk/download.html`. If you check out the source via CVS, you end up with top-level source directories named `tcl` and `tk`. In contrast, if you download a specific released version of the source, the top-level directories have the version number appended to their names, such as `tcl8.5.5`.

Each source distribution has a `README` file in its top-level directory with general information and references to additional information about Tcl. There is also a set of subdirectories, including

- `doc/`—Tcl/Tk reference documentation
- `generic/`—source code applicable to all supported platforms
- `library/`—a library of Tcl scripts used by Tcl/Tk
- `macosx/`—Macintosh-specific source code, make files, and Xcode project files
- `tests/`—the Tcl/Tk test suite
- `tools/`—tools used when generating Tcl distributions
- `unix/`—Unix-specific source code, configure, and make files
- `win/`—Windows-specific source code and make files for use with

Microsoft Visual C++

Each significant subdirectory includes its own `README` file describing its contents. The `README` files for the `macosx`, `unix`, and `windows` directories also provide up-to-date information for building Tcl or Tk for those platforms. You should always consult these `README` files to check for any new dependencies, system requirements, or build procedures.

47.1.1 Building Tcl and Tk on Unix

Building Tcl on Unix is fairly straightforward. Tcl uses GNU Autoconf to handle system build dependencies. Therefore, you can configure, build, and install Tcl or Tk using the typical procedure:

```
# cd to tcl/unix or tk/unix
./configure
make
make install
```

Note

Before doing the `make install`, you can optionally run `make test` to run the Tcl or Tk test suite to verify the build on your system. Be aware that the test suite might take considerable time to execute.

You can provide standard options to `configure`, such as `--prefix` to specify the installation directory and `--exec-prefix` to specify the installation directory for architecture-specific files. The `--prefix` option is especially handy for installing multiple versions of Tcl on a single system or building one or more personal Tcl/Tk installations for testing purposes without interfering with the system version of Tcl/Tk. The default prefix on Unix systems is `/usr/local`, although Tcl is also often installed (for example, on Linux) in `/usr`. See the Autoconf documentation for more information on standard `configure` options (http://www.gnu.org/software/autoconf is one online resource).

Additionally, Tcl and Tk provide some package-specific options to `configure` that affect how they are compiled. Most of them follow the standard syntax of enabling or disabling a feature using the `--enable-`*`feature`* and `--disable-`*`feature`* options. The more significant ones include

- `--enable-shared`

Compile as a shared library if enabled (the default), otherwise as a static library.

- `--enable-threads`

Compile with multithreading support if enabled; the default is disabled. (Many precompiled distributions of Tcl now enable this feature.)

- `--enable-64bit`

Enable 64-bit support (where applicable); the default is disabled.

- `--enable-symbols`

Compile with debugging symbols if enabled; the default is disabled.

- `--with-tcl=`*`location`*

(Tk only) Specifies the directory where Tcl was built. By default, the Tk build system assumes that this directory is `../../tcl`*`<version>`*, where *`<version>`* is the same version of Tk being built. If this is not the case, you must specify the appropriate directory with `--with-tcl` to build Tk successfully.
Among other things produced while building Tcl is a file called `tclConfig.sh`, which is used by the Tcl Extension Architecture (described in Section 47.2) to hold information about the Tcl installation for use with extension compilation. It is a shell script that defines a number of variables such as the Tcl version; the C compiler used to compile Tcl; compiler and linker flags; the location of include files, libraries, and paths; and so on. Building Tk creates an analogous file, `tkConfig.sh`.

47.1.2 Building Tcl and Tk on Mac OS

Building Tcl on the Macintosh requires at least Mac OS X 10.1; earlier versions are no longer supported. (Tcl/Tk 8.4 was the last version that had support for the "classic" Mac OS series.) You must also have Apple's Developer Tools installed on your system.
One option for building Tcl/Tk on the Macintosh is to use the Unix build system found in `tcl/unix` and `tk/unix`. In that case, you follow the exact same steps as described in Section 47.1.1. There are a few additional `configure` options that apply only to builds on Mac OS:

- `--enable-corefoundation`

Use Apple's Core Foundation framework; enabled by default.

- `--enable-framework`

Package shared libraries in Mac OS X frameworks; disabled by default.

- `--enable-aqua`

(Tk only) Use the Aqua windowing system, rather than X11; disabled by default. Requires Tcl and Tk to be built with the Core Foundation

framework.
Alternatively, the `tcl/macos` and `tk/macos` directories contain a `GNUmakefile` that you can use to build and install Tcl and Tk directly. The following steps are sufficient for a standard installation. They assume that your Tcl and Tk source directories are contained within a common parent, and for generality, they're presented assuming that a shell variable name `ver` exists containing the version string (e.g., `8.5.5`) for a specific released version, or an empty string if you are building from CVS. First, to actually build the packages, you would open a terminal, go to the common parent directory, and execute

```
make -C tcl${ver}/macosx
make -C tk${ver}/macosx
```

Then, to install the package onto your system's root volume (which requires the administrator password):

```
sudo make -C tcl${ver}/macosx install
sudo make -C tk${ver}/macosx install
```

To install into an alternate location, such as your home directory (for example, if you don't have administrator privileges, or if you want to test against multiple versions of Tcl):

```
make -C tcl${ver}/macosx install INSTALL_ROOT="${HOME}/"
make -C tk${ver}/macosx  install INSTALL_ROOT="${HOME}/"
```

Other `make` targets and options are available. Consult the `README` file for more information.
Finally, the `macosx` subdirectories also contain several project files that you can use with different versions of Apple's Xcode integrated development environment. Once again, see the `README` file for information on which project file to use and how the files are configured.

47.1.3 Building Tcl and Tk on Windows

There are several options for building Tcl/Tk on Windows, depending on what tools you have installed.
If you have Microsoft Visual C++, the `tcl/win` and `tk/win` directories contain a `makefile.vc` file that you can use with `nmake`. Each file contains comments describing the targets and options available. The `tcl/win` directory contains a Microsoft Developer Studio workspace and project file that you can use as

well.
Alternatively, you can use the open-source MinGW and MSYS tools to build Tcl on Windows. (See http://www.mingw.org to read more about and to download these tools.) These tools provide a “Minimalist GNU for Windows,” including standard GNU compilers and build utilities. For this option, `tcl/win` and `tk/win` include `configure` scripts that you can use to follow the same sequence of `configure`, `make`, and `make install` as for Unix. If you choose this approach, you can use the same set of `configure` options as described in Section 47.1.1.

47.2 The Tcl Extension Architecture (TEA)

If you are writing your own extension in C, you could use any build system you like to compile the extension for use, write your own make files, and so on. However, if you plan to distribute your extension for use on a variety of platforms, it’s very difficult to design a general system to handle all of your target platforms.
Tcl has a standard system for building extensions called TEA, which stands for “Tcl Extension Architecture.” TEA is based on the GNU Autoconf tool and provides a system for building extensions. TEA also defines best practice policies for extension development and distribution.
The starting point for using TEA for your extensions is a sample extension designed to illustrate the TEA system and to serve as a template for extension development. The sample extension is maintained on SourceForge as part of the Tcl project (http://tcl.sourceforge.net). You can obtain it from the CVS repositories via anonymous access as described on the site. The module name is `sampleextension`.

Note

The sample extension also contains a subdirectory named `tea`, which contains more detailed documentation on TEA as well as guidelines for developing portable extensions. As TEA continues to evolve, this is likely to be the most up-to-date documentation on TEA.

The heart of TEA is Autoconf and the `configure` script that it generates. Autoconf takes two template files, `configure.in` and `Makefile.in`, along with a set of M4 macro definitions, supplied in a file named `aclocal.m4`, and generates an appropriate `configure` script that people can use to build your extension. The sample extension ships with commented, general-purpose `configure.in` and `Makefile.in` files that you can adapt for use by your extension, as well as a set of M4 macros to include in your `aclocal.m4` that implement the TEA build infrastructure. In practice, if you start with the files from the sample extension, all you need to do is edit the `configure.in` and `Makefile.in` files, then execute `autoconf` in the directory containing your extension to generate your extension's `configure` script.

Note

In addition to the sample extension, TEA assumes that you have development tools installed on your system, including Autoconf. To use Autoconf to create a `configure` script on a Windows development system, you must have the open-source MinGW and MSYS tools installed as well as Autoconf. (See http://www.mingw.org to read more about and to download these tools.)

Autoconf is required only for your development system so that you can create the `configure` script for your extension. Users of your extension, who would build it on their own systems, require only appropriate development tools to be installed. They can then use the `configure` script you provide to actually build the extension on their systems.
After the `configure` script has been run, your extension should be ready to build, which you can do by issuing the `make` command. Installation is handled by `make install`. Other targets include `make test`, if you have defined tests and provided a way to run them, and the extremely useful `make dist`, which bundles your source and `configure` files into a distribution (by default, a gzipped tar file) that is ready to ship.

47.2.1 TEA Standard Configure Options

TEA defines a large set of options for the `configure` script it generates, which users of your extension can use to configure your extension when they build it on their systems. Most of them follow the standard syntax of enabling or disabling a feature using the `--enable-`*`feature`* and `--disable-`*`feature`* options. The more significant ones include

- `--enable-shared`

Compile as a shared library if enabled (the default), otherwise as a static library.

- `--enable-threads`

Compile with multithreading support if enabled (the default).

- `--enable-64bit`

Enable 64-bit support (where applicable); the default is disabled.

- `--enable-symbols`

Compile with debugging symbols if enabled; the default is disabled.

- `--with-tcl=`*`location`*

Specifies the directory where Tcl was built. This allows you to build your extension for multiple versions of Tcl or for multiple platforms. If you don't have multiple versions of Tcl installed on your system, Autoconf can often find this information automatically.

- `--with-tk=`*`location`*

Specifies the directory where Tk was built. This option might be required only if your extension uses the Tk C API. If you don't have multiple versions of Tk installed on your system, Autoconf can often find this information automatically.

- `--with-tclinclude=`*`location`*

Specifies the directory where the Tcl include files can be found (most notably, `tcl.h`).

- `--with-tcllib=`*`location`*

Specifies the directory where the Tcl libraries can be found (most notably, `libtclstubs.a`).

- `--prefix`

Defines the root of the installation directory. Defaults to the value given to Tcl when it was built.

- `--exec-prefix`

Defines the root of the installation directory for platform-specific files. Defaults to the value given to Tcl when it was built.

47.2.2 Directory Layout for TEA Extensions

TEA recommends a standard directory structure for organizing your extension's source within a common parent directory:

- `demos/`—demonstration scripts for the extension
- `doc/`—extension reference documentation
- `generic/`—source code applicable to all supported platforms

- `library/`—supporting scripts, such as default bindings for widgets
- `macosx/`—Macintosh-specific source code
- `tests/`—the extension test suite
- `unix/`—Unix-specific source code
- `win/`—Windows-specific source code

Additionally, the sample extension provides the following files and directories in support of TEA:

- `tclconfig/`—a directory containing `tcl.m4`, which defines a collection of Autoconf macros used by TEA, and `install-sh`, a program used for copying files to their installation locations
- `tea/`—a directory containing documentation on the current version of TEA
- `aclocal.m4`—a file used by Autoconf as input when generating the final `configure` script
- `configure.in`—the script template used by Autoconf to generate the final `configure` script
- `Makefile.in`—the template used by `configure` to generate the final `Makefile`
- `pkgIndex.tcl.in`—the template used by `configure` to generate the final `pkgIndex.tcl` file

The `make install` command installs a TEA package relative to the `--prefix` and `--exec-prefix` directories. For example, for the compiled library files and `pkgIndex.tcl` file, the installer creates a subdirectory under the *`$prefix`*`/lib` directory consisting of the package name concatenated with the package version number, such as `thread2.6.5`. By default, files are installed to the following locations:

- *`$exec-prefix`*`/lib/`*`$package$version`*`/`—compiled library files and the `pkgIndex.tcl` file
- *`$exec-prefix`*`/bin/`—binary executables and dependent `.dll` files on Windows
- *`$prefix`*`/include/`—C header files
- *`$prefix`*`/man/`—Unix man pages from the `doc` source directory

47.2.3 Customizing the `aclocal.m4` File

One input to Autoconf is the `aclocal.m4` file, located in the root of your source directory. This file defines the M4 macros used to process the `configure.in` file and produce the `configure` script. The sample extension provides the file `tclconfig/tcl.m4` that defines all of the M4 macros needed for the TEA build system, so in most cases your `aclocal.m4` file can contain just the following line:

```
builtin(include,tclconfig/tcl.m4)
```

If you want to extend the `configure` script options for your extension or otherwise customize the build system, you can add more macro definitions to this file.

47.2.4 Customizing the `configure.in` File

The `configure.in` file is a template containing a series of M4 macros that determine how the resulting `configure` script behaves. The sample extension provides a well-annotated version of this file, and in most cases you can simply edit the file as needed in the sections marked for you to change. In general, the order of the macros is important; mixing them up can cause problems that are difficult to trace.
The first macro in your `configure.in` file is `AC_INIT`, which initializes the environment and specifies the name and version number of your extension:

```
AC_INIT([tclifconfig], [0.1])
```

Next you initialize the TEA variables, specifying the version of TEA, and instruct Autoconf to use the `tclconfig` subdirectory for additional build scripts and definitions:

```
TEA_INIT([3.7])
AC_CONFIG_AUX_DIR(tclconfig)
```

The next two macros are responsible for finding and loading the `tclConfig.sh` file, which is created when you build Tcl. It contains information such as the Tcl version; the C compiler used to compile Tcl; compiler and linker flags; the location of include files, libraries, and paths; and so on. TEA incorporates heuristics to try to find a Tcl installation on the target system containing a `tclConfig.sh` file, but a person building your extension can also use the `--with-tcl` option to `configure` to identify a specific directory. This is especially useful when building your extension against multiple versions of Tcl or for multiple platforms. There are no arguments to provide to these macros:

```
TEA_PATH_TCLCONFIG
TEA_LOAD_TCLCONFIG
```

If your extension also uses the Tk C API, you need two macros to find and load an analogous `tkConfig.sh` file created when Tk is built:

```
#TEA_PATH_TKCONFIG
#TEA_LOAD_TKCONFIG
```

The next two macros are required to handle the `--prefix` argument and to identify the appropriate C compiler and its options:

```
TEA_PREFIX
TEA_SETUP_COMPILER
```

The next set of macros is specific to your extension. Most important is a space-separated list of C source files to compile; you do not need to specify the subdirectory containing the file as long as it is one of the standard TEA source directories:

```
TEA_ADD_SOURCES([sample.c tclsample.c])
```

If other extensions or applications would compile against your extension, you might need to provide a list of public header files that you want installed on the target system with the `make install` command:

```
TEA_ADD_HEADERS([])
```

If required, you can list additional libraries needed to compile the extension, additional directories containing required include files, and additional compiler flags:

```
TEA_ADD_LIBS([])
TEA_ADD_INCLUDES([])
TEA_ADD_CFLAGS([])
```

You can also provide a list of additional Tcl source files that are part of your extension:

```
TEA_ADD_TCL_SOURCES([])
```

In the next section, you can provide platform-specific configuration instructions. In the condition sections, you can include additional calls to the `TEA_ADD_*` macros to list platform-specific source files, dependent libraries, and so forth:

```
if test "${TEA_PLATFORM}" = "windows" ; then
    ....
else
    ....
fi
```

Next you specify which Tcl and Tk headers your extension needs. At a minimum, you'll need the Tcl public header (`tcl.h`). If you need to compile against the Tk C API, you'll also need the Tk public header (`tk.h`) and the macro for locating the X11 headers. If at all possible, you should avoid using the private headers because the data structures and API may change without notice, whereas the Tcl Core Team makes every effort to keep the exported public C API very stable:

```
TEA_PUBLIC_TCL_HEADERS
#TEA_PRIVATE_TCL_HEADERS
#TEA_PUBLIC_TK_HEADERS
#TEA_PRIVATE_TK_HEADERS
#TEA_PATH_X
```

The next set of macros does some standard processing during configuration—checking whether or not to compile with multithreaded support, to build shared or static libraries, to include symbols or not, and general compiler options:

```
TEA_ENABLE_THREADS
TEA_ENABLE_SHARED
TEA_CONFIG_CFLAGS
TEA_ENABLE_SYMBOLS
```

Chapter 36 discussed the use of Tcl's stubs mechanism, which allows an extension compiled against one version of Tcl to be used with later versions of Tcl. Extensions should almost always use stubs to be as independent as possible of Tcl versions. On the other hand, if you are compiling code that links to Tcl as a library, rather than creating an extension, you should not use stubs—they serve no purpose. Use the following declaration to enable stubs for the Tcl C API and, if needed, the Tk C API:

```
AC_DEFINE(USE_TCL_STUBS)
#AC_DEFINE(USE_TK_STUBS)
```

Next is a standard TEA macro that generates a command line to use when building a library:

```
TEA_MAKE_LIB
```

The following macros determine the names of the `tclsh` and/or `wish` executables, which are used by the build system only to run test cases in response to the `make test` command:

```
TEA_PROG_TCLSH
#TEA_PROG_WISH
```

The last line of the file lists all of the files that the `configure` script needs to create:

```
AC_OUTPUT([Makefile pkgIndex.tcl])
```

Each file listed must have a template file in the same directory, named with a `.in` suffix. At a minimum, you'll need `Makefile.in`. You can also have Autoconf manage the creation of a `pkgIndex.tcl` file, as illustrated by the sample extension. For a cross-platform extension, you might find it easier to maintain separate make files in the different source subdirectories, such as the following code that creates additional make files in the `generic`, `unix`, and `win` subdirectories from the respective templates:

```
AC_OUTPUT([
Makefile
generic/Makefile
unix/Makefile
win/Makefile
])
```

Some additional TEA macros are available which are not covered in this section, and you can also use standard Autoconf macros to manage aspects of the build process. See the TEA documentation that comes with the sample extension for more information on these features.

47.2.5 Customizing the `Makefile.in` File

The `Makefile.in` file serves as a template for the `Makefile` that is to be generated. Trying to create a `Makefile.in` from scratch that you could use with TEA would be difficult and tedious. The recommended approach is to base your make file template on the `Makefile.in` that comes with the sample extension, which in most cases works "out of the box." You might want to customize it to go beyond the basic build process, such as using it for running tools to build your documentation in different formats on different platforms (e.g., Unix man pages versus Windows help files) or performing custom cleanup operations. The actual customization of the file is beyond the scope of this chapter. See the TEA documentation that comes with the sample extension for more information.

47.2.6 Building an Extension on Windows

TEA is based on Autotools, the GNU build system. The open-source MinGW and MSYS tools provide the environment necessary to build a TEA extension on a Windows system (see http://www.mingw.org). If you feel that your target users are willing to use MinGW and MSYS to build your extension on Windows, you can use the same `configure` script as for Unix; Windows users would do the same sequence of `configure`, `make`, and `make install` to build your extension on their systems.
If you prefer, you can decide to create and deliver a Visual C++ make file for your extension as well. The sample extension provides an example in `win/makefile.vc`. It has been designed to be as generic as possible but still requires some additional maintenance to synchronize with the TEA `configure.in` and `Makefile.in` files. Instructions for using the Visual C++ make file are written in the first part of the `makefile.vc` file.

47.3 Building Embedded Tcl

When Tcl is embedded in other systems, it is probably necessary to integrate the Tcl build system with that of your program. If the program that embeds Tcl uses Autotools, it shouldn't be too difficult to accomplish. The basic approach is to use the TEA macros by including the `tcl.m4` file, so that it's possible to obtain information about how Tcl is compiled.

A. Installing Tcl and Tk

The source code and reference documentation for Tcl and Tk are freely available. Tcl and Tk are available under a BSD-style license, which provides great flexibility, including the use of Tcl and Tk in commercial products without royalties or being required to open-source your own code. This appendix describes different methods for retrieving the Tcl and Tk distributions for use on your local system.

A.1 Versions

Tcl and Tk are distributed separately, and each distribution has a version number. A version number consists of two or more integers separated by a period, such as 8.5.3 or 8.4. The first of the numbers is the *major version number* and the second is the *minor version number*. A third number, if present, indicates a *patch release*, usually containing only bug fixes; it's rare for a patch release to introduce any significant new functionality. Each new release of Tcl or Tk gets a new version number. If the release is compatible with the previous release, it is called a *minor release*: the minor version number increments and the major version number stays the same. For example, Tcl 8.5 was a minor release that followed Tcl 8.4. Minor releases are compatible in the sense that C code and Tcl scripts written for the old release should also work under the new release without modifications. If a new release includes incompatible changes, it is called a *major release*: the major version number increments and the minor version number resets to 0. For example, Tcl 8.0 was a major release that followed Tcl 7.6. When you upgrade to a major release, you will probably have to modify your scripts and C code to work with the new release.

Although Tcl can be used by itself, each release of Tk is designed to be used with a particular release of Tcl. When it starts up, Tk checks the Tcl release number to be sure that it will work with Tk. The matching releases of Tcl and Tk don't necessarily have the same version numbers; for example, Tk 3.6 required Tcl 7.3. However, all modern Tcl and Tk releases since 8.0 share version numbers to avoid confusion.

A.2 Bundled Tcl Distributions

Many Unix and Unix-like operating systems have precompiled Tcl/Tk distribution packages available. (Many have the Tcl and Tk source available as distribution packages as well.) Some operating systems include Tcl and Tk by default when you install the operating system or offer it as an option in a "developer" package. If Tcl and Tk are not available on your operating system's installation media, or you want to upgrade to a newer version of Tcl/Tk, you should be able to find Tcl/Tk distribution packages from the official package repositories for your operating system. You can use whatever package management utilities are supported on your operating system (for example, `rpm`, `yum`, `dpkg`, `apt-get`, etc.) to handle installing, upgrading, and uninstalling Tcl/Tk on your system.
One disadvantage to using such official Tcl/Tk distribution packages is that there can be a delay—sometimes substantial—between the release of a new version of Tcl/Tk and the availability of an official distribution of the new version for your operating system. It can also be more difficult to install multiple versions of Tcl/Tk for use on the same system, or to install and maintain Tcl/Tk for multiple platforms for use from a shared network file system.

A.3 ActiveTcl

ActiveState Software (`http://www.activestate.com`) is a company that creates development tools and provides service and support for several dynamic languages, including Tcl. In addition to their commercial products, ActiveState offers a free, precompiled version of Tcl called ActiveTcl, which also bundles several popular Tcl extensions. ActiveTcl is available for several platforms, including Windows, Linux, Mac OS X, Solaris, AIX, and HP-UX at this time. ActiveTcl is often the easiest way to install Tcl/Tk on your system if your focus is Tcl/Tk script development.

A.4 Tclkits

Chapter 14 discussed the use of Tclkits as a strategy for distributing your Tcl/Tk-based applications. A Tclkit is a single-file executable containing the entire Tcl distribution in a remarkably small footprint (typically well under 2MB). You can use the Tclkit executable as a Tcl shell, just as you would `tclsh`. Precompiled Tclkits are available for a variety of platforms. You can find out more about Tclkits at http://www.equi4.com/tclkit and http://wiki.tcl.tk/tclkit and download Tclkits from http://www.equi4.com/tclkit/download.html.

A.5 Compiling Tcl/Tk from Source Distributions

The source code for Tcl and Tk is maintained on SourceForge, whose main project home page is at http://tcl.sourceforge.net. Tcl and Tk are maintained as separate projects on the site. You can download released versions of the source code from the file distribution links, or you can access the CVS repositories via anonymous access as described on the site. Released versions of the sources are also available from http://www.tcl.tk/software/tcltk/download.html. Detailed instructions for compiling and installing Tcl and Tk are provided in Chapter 47.

B. Extensions and Applications

Tcl/Tk has an active user community. Many people have built packages that extend the base functionality of Tcl and Tk and applications based on Tcl and Tk. Several of these packages and applications are publicly available and widely used in the Tcl/Tk community. There isn't space in this book to discuss all of the available Tcl/Tk software in detail, but this appendix gives a quick overview of several extensions and applications.

B.1 Obtaining and Installing Extensions

The best source for information on available Tcl extensions and applications is the Tcler's Wiki (`http://wiki.tcl.tk`). There you can search for topics of interest and find links to related extensions and applications. The Wiki also contains a page listing Tcl/Tk extensions, `http://wiki.tcl.tk/940`. Another excellent resource is the "Great Unified Tcl/Tk Extension Repository" maintained by Joe English, `http://www.flightlab.com/~joe/gutter/browse.html`. Each provides links to where you can download the desired extensions. Authors also often announce new versions of extensions and applications on the `comp.lang.tcl` Usenet newsgroup (see Section C.1).

B.1.1 Installing Extensions Manually

Once you have downloaded an extension, follow the instructions that accompany it for installing it on your system. Some extensions provide installer applications, and some you copy to an appropriate location where Tcl can find them on your system. A README file should be included with the extension that tells how to install it and describes any peculiarities.

If the extension provides no installation instructions, usually you can simply copy the extension to one of the directories where Tcl looks for extensions by default. If the extension is distributed as a Tcl module, you can copy the file to one of the directories described in Section 14.6.2. If the extension is distributed as a package with separate files, including a `pkgIndex.tcl` file, you can move the directory containing the extension files to one of the directories described in Section 14.5.4.

B.1.2 Installing Extensions from ActiveState TEApot Repositories

ActiveState (`http://www.activestate.com`) has developed a Tcl package management system that provides a simple method for obtaining, installing, and updating Tcl extensions and applications. It allows you to use a client application named `teacup` to browse, search, and install extensions from hosted repositories running a repository server application named `teapot`. ActiveState's free ActiveTcl distribution includes the `teacup` application and documentation, but you can also download a copy of it for your platform from `http://teapot.activestate.com`. ActiveState hosts a public TEApot at `http://teapot.activestate.com` containing a variety of extensions and applications. You can get help for `teacup` and its options by executing `teacup help`. Additionally, you can read more about `teacup` on the Tcler's Wiki at `http://wiki.tcl.tk/teacup`. The documentation included with ActiveTcl also describes the availability of `teapot` for establishing your own TEApot repositories (e.g., a company-hosted TEApot containing approved versions of open-source and proprietary extensions for internal use).

The `teacup` application includes a set of subcommands for managing one or more local *installation repositories* that serve as a source of extensions for use by your Tcl installation. In other words, if you have installed a particular extension in one of your installation repositories, the Tcl shells configured to use the repositories can successfully load the extension with a `package require` command. Most often you would use just one installation repository, known as the *default installation repository*. You can query or change the default repository directory with the `teacup default` command:

```
teacup default
⇒ /Library/Tcl/teapot
```

You can also create and delete additional repositories and configure your Tcl installations to use different sets of repositories. See the `teacup` documentation for more details.

The `teacup list` command lists the extensions available from the TEApot. You can also provide a specific extension name and optionally a version of that extension. If there is not an exact match for the name, the command does a case-insensitive substring search for candidate names. To list extensions that you have installed in your default repository, include the `--at-default` option; for example:

```
   teacup list --at-default file
⇒ entity  name     version platform
   ------- -------- ------- --------
   package fileutil 1.13.4  tcl
   ------- -------- ------- --------
   1 entity found
```

You can get a description of an extension using the `teacup describe` command. As with `teacup list`, you can get descriptions either for extensions available on the TEApot or, with the `--at-default` option, for installed extensions; for example:

```
   teacup describe Img
⇒ Entity        Img

   Description @ http://teapot.activestate.com
   The Img package provides support for several image formats
   beyond the standard formats in Tk (PBM, PPM, and GIF),
   including BMP, XBM, XPM, GIF (no LZW), PNG, JPEG, TIFF,
   and PostScript.
```

The `teacup install` command installs a named extension. You can optionally specify a minimum or exact version to install; by default the latest version available is installed. If the extension has dependencies on other extensions not already installed, `teacup` automatically installs them as well. You can use the `--dry-run` option to simulate the installation, in which case `teacup install` reports the steps it would take during the installation.
To upgrade your installation repository, use the `teacup update` command. By default, this updates all installed extensions and applications to the latest version available in the TEApot *and* installs any uninstalled extensions and applications from the TEApot. You can use the `--only newer` option to upgrade only currently installed extensions, or the `--only uninstalled` option to install extensions and applications you don’t have installed already. The `--dry-run` option is also available to simulate the update process.
The `teacup remove` command uninstalls a named extension or application. If you invoke it without naming an extension or application, it removes all items from the installation repository. The `--dry-run` option for simulating removal is also available for this command.
You can use the `teacup version` command to determine what version of `teacup` is installed. To upgrade to the latest version of `teacup`, execute `teacup upgrade-self`.

Note

If you want to use `teacup` with a Tcl distribution other than ActiveTcl, you can download a copy of it for your platform from http://teapot.activestate.com. You then need to set up an installation repository if you don't already have one, set up your Tcl shell to be able to use repositories, and then link your shell to the repository you created. You can accomplish this with the following set of commands:

```
teacup create /path/to/your/repository
teacup setup /path/to/your/shell
teacup link make /path/to/your/repository /path/to/your/shell
```

B.2 TkCon Extended Console

TkCon is an enhanced console for interacting with Tcl. Although the `wish` interpreter has a bare-bones console, TkCon extends this basic functionality with many useful facilities to aid Tcl programming, including

- Interactive command editing
- A cross-session persistent command history
- Enhanced history searching
- Command, variable, and path expansion
- Electric character matching (similar to `emacs`)
- Command and variable highlighting
- Capture of input, output, error, or command history to log files
- "Hot errors," where clicking on an error result displays a stack trace
- Interactive debugging features
- Extensive configuration options

TkCon, developed by Jeff Hobbs, is written entirely in Tcl/Tk and can be embedded into a Tcl/Tk application to provide console support for the application. For more information, see the TkCon SourceForge page (http://tkcon.sourceforge.net) and the TkCon page on the Tcler's Wiki (http://wiki.tcl.tk/tkcon).

B.3 The Standard Tcl Library, Tcllib

Tcllib, otherwise known as the Standard Tcl Library, contains several independent packages providing a variety of commonly used features, including

- Networking protocol implementations, including e-mail (POP3 and SMTP), domain name resolution (DNS), file transfer (FTP), directory (LDAP), Usenet news (NNTP), chat (IRC), and network time (NTP)
- Web support for parsing and generating HTML, generating JavaScript, writing CGI scripts, and working with URLs and MIME attachments
- Encryption and checksum support for CRC checksums, DES, MD4, MD5, SHA1, and base64 encoding
- Programming utilities for logging, profiling performance, and generating documentation
- Data structure implementations, including stacks, queues, matrixes, graphs, and trees
- The Snit object-oriented framework
- Text-processing utilities, including CSV processing
- Advanced math functions

For more information, see the Tcllib SourceForge page (http://tcllib.sourceforge.net) and the Tcllib page on the Tcler's Wiki (http://wiki.tcl.tk/tcllib).

B.4 Additional Image Formats with Img

Img is a Tk extension that adds support for many image formats beyond the small number supported natively by Tk. Img includes support for formats including BMP, GIF, ICO, JPEG, Pixmap, PNG, PPM, PostScript, SGI, Sun, TGA, TIFF, XBM, and XPM. The Img extension comes from Jan Nijtmans. For more information, see the Img page on the Tcler's Wiki (http://wiki.tcl.tk/img) and the Img SourceForge page (http://sourceforge.net/projects/tkimg).

B.5 Sound Support with Snack

The Snack sound extension adds commands to play and record audio. Snack supports in-memory sound objects, file-based audio, and streaming audio, with background audio processing. It handles file formats such as AIFF, AU, MP3, NIST/Sphere, and WAV. Snack is courtesy of Kåre Sjölander. You can find out more about Snack at the Snack home page (http://www.speech.kth.se/snack) and the Snack page on the Tcler's Wiki (http://wiki.tcl.tk/snack).

B.6 Object-Oriented Tcl

One of the historical criticisms of Tcl has been the lack of native object-oriented structures. However, this resulted in people developing several object-oriented extensions, each with its own unique features and strengths.
[incr Tcl], also known as Itcl (the name is a play on the Tcl version of the C++ operator used to name C++), implements a framework very similar to C++. You can define classes consisting of data members and methods, with public, protected, and private protection levels available for each. Inheritance, including multiple inheritance, is supported. Additionally, you can use the object-oriented Tk widgets in the [incr Tk] and [incr Widgets] packages. [incr Tcl] was created by Michael McLennan. For more information, see the [incr Tcl] page on the Tcler's Wiki (http://wiki.tcl.tk/62) and the [incr Tcl] SourceForge page (http://incrtcl.sourceforge.net/itcl).
XOTcl, or extended object Tcl, extends an earlier object-oriented package called OTcl. XOTcl provides a number of object-oriented concepts to help reduce the complexity of large-scale programs. These concepts include mix-in classes, nested classes, aggregations, forwarders to support delegation, slots for storing data, and filters that provide features similar to aspect-oriented programming. XOTcl comes from Uwe Zdun and Gustaf Neumann. For more on XOTcl, see the XOTcl home page (http://www.xotcl.org) and the XOTcl page on the Tcler's Wiki (http://wiki.tcl.tk/xotcl).
Snit, short for Snit's Not Incr Tcl, provides an alternative object-oriented extension written in pure Tcl. Unlike most object-oriented languages or language extensions, Snit is based on delegation, not inheritance. Snit objects delegate work to child components, rather than inherit from base classes. While it used to be a separate extension, Snit is now part of Tcllib, described in Section B.3. Snit was developed by William Duquette. For more on Snit, see the Snit page on the Tcler's Wiki (http://wiki.tcl.tk/3963) and the Snit reference page (http://tcllib.sourceforge.net/doc/snit.html).

Note

Tcl 8.6 will introduce a native object-oriented framework. Although it will be sufficient for stand-alone use in scripts, its primary purpose is to serve as a core for other object-oriented extensions, making them easier to implement and more efficient.

B.7 Multithreaded Tcl Scripting

Chapter 47 discussed the requirements for embedding Tcl in a multithreaded application and described the functions in Tcl's C API that support platform-neutral, multithreaded development. Tcl does not have any built-in commands that expose multithreading at the script level. The Thread extension—originally written by Brent Welch and now maintained by Zoran Vasiljevic—provides a set of script-level commands for creating and managing threads in an application.

The Thread extension supports what is often called an *apartment threading model*. Each thread created at the script level has its own Tcl interpreter, which maintains its own variables, procedures, namespaces, and other state information. The Thread extension also implements a message-passing mechanism for inter-thread communication, shared variables, mutual exclusions (mutexes), condition variables, and thread pools.

You can use the Thread extension only in a `tclsh`, `wish`, or other application that has been compiled with multithreaded support enabled, as described in Chapter 46. Also, any other binary extensions used by your application must be compiled with multithreaded support enabled as well.

The source code for the Thread extension is maintained as a project on the Tcl SourceForge site: `http://tcl.sourceforge.net`. You can read more about the Thread extension on the Tcler's Wiki, at `http://wiki.tcl.tk/thread`.

B.8 XML Programming

XML programming has become more important over the years, as XML has become a common serialization and data exchange format. Tcl has several extensions related to XML processing.

TclXML groups a number of extensions related to parsing XML documents. The base extension, TclXML, handles parsing XML documents. TclDOM adds a document object model, a way to manipulate XML as a tree structure in memory. TclXSLT adds XSL transformations, and TclTidy cleans up XML and HTML and is particularly useful for parsing HTML from web pages, many of which do not truly conform to HTML standards. TclXML was created by Steve Ball. For more information, see the TclXML SourceForge page (http://tclxml.sourceforge.net) and the TclXML page on the Tcler's Wiki (http://wiki.tcl.tk/tclxml).

tDOM provides an alternative for processing XML documents, including XSLT support. It is known for its processing speed and low memory consumption. The tDOM project was started by Jochen Loewer and is currently maintained by Rolf Ade. You can find out more about it at the tDOM home page (http://www.tdom.org) and the tDOM page on the Tcler's Wiki (http://wiki.tcl.tk/tclxml).

TclSOAP helps you create XML messages for SOA, or service-oriented architectures, using the Simple Object Access Protocol, or SOAP. With SOAP, applications send and receive XML messages; the major advantage is that you can write programs in any language, such as Tcl, and not worry about the implementation of the program on the other side. TclSOAP was created by Pat Thoyts. For more information, see the TclSOAP SourceForge page (http://tclsoap.sourceforge.net) and the TclSOAP page on the Tcler's Wiki (http://wiki.tcl.tk/tclsoap).

B.9 Database Programming

Although Tcl doesn't support database access natively, you can add a number of extensions to help access data from relational databases.

Oratcl allows Tcl applications to access Oracle databases (http://oratcl.sourceforge.net). Mysqltcl similarly provides an interface to MySQL databases (http://www.xdobry.de/mysqltcl). PostgreSQL has the pgtcl extension for database access (http://pgfoundry.org/projects/pgtcl). Sybtcl provides Sybase support (http://sybtcl.sourceforge.net). The sqlite2 extension provides access to SQLite (http://www.sqlite.org). TclODBC allows Tcl applications to call on ODBC database drivers, most commonly used on Windows (http://sourceforge.net/projects/tclodbc).

Note

> Tcl 8.6 will introduce TDBC, an extension shipped as part of the Tcl core, to provide a standard interface for SQL database access. For more information, see the TDBC page on the Tcler's Wiki (http://wiki.tcl.tk/tdbc).

Another popular option is Metakit, which is designed to be an efficient embedded database library with a small footprint. The Mk4tcl extension provides a Tcl API for accessing Metakit. See http://www.equi4.com/metakit for more information.

B.10 Integrating Tcl and Java

The Tcl/Java Project provides two routes for integrating Tcl and Java code. One option is TclBlend, an extension that supports loading a Java interpreter into an existing Tcl process. As a result, you can use Tcl script commands to load Java classes, create objects, invoke methods, and so forth. The other option is Jacl, a self-contained implementation of a Tcl interpreter written entirely in Java. The intent of Jacl is to incorporate scripting functionality into an existing Java application. You can find out more at the Tcl/Java home page (http://tcljava.sourceforge.net) and the Tcl/Java page on the Tcler's Wiki (http://wiki.tcl.tk/tcljava).

B.11 SWIG

If you have an existing library of C/C++ code, you might want to make its functionality available through Tcl script commands. Although you could write your own wrapper code to expose the various functions as Tcl commands, a tool called SWIG can create the wrapper code for you automatically. SWIG is not limited to use with just Tcl but can also generate wrapper code for many other dynamic languages. See the SWIG home page (http://www.swig.org) for more information.

B.12 Expect

Expect is one of the oldest Tcl applications and also one of the most popular. It is a program that "talks" to interactive programs. An Expect script describes what output can be expected from a program and what the correct responses should be. It can be used to automatically control programs such as `ftp`, `telnet`, `ssh`, `fsck`, and others that cannot be automated from a shell script because they require interactive input. Expect also allows the user to take control and interact directly with the program when desired. For example, the following Expect script logs into a remote machine using the `ssh` program, sets the working directory to that of the originating machine, then turns control over to the user:

```
package require Expect
spawn ssh [lindex $argv 0]
expect -nocase "password:"
send [lindex $argv 1]
expect -re "(%|#) "
send "cd [pwd]\r"
expect -re "(%|#) "
interact
```

The `spawn`, `expect`, `send`, and `interact` commands are implemented by Expect, but `lindex` and `pwd` are built-in Tcl commands. The `spawn` command starts up `ssh`, using a command-line argument as the name of the remote machine (`lindex` extracts the first argument from the command line, which is available in the variable `argv`). The `expect` command waits for `ssh` to output a password prompt, matching it in a case-insensitive manner. The script assumes that the password is the second command-line argument and sends it to the `ssh` program. (This approach is insecure and shown for simplicity only. There are several other approaches for handling password prompts like this in Expect that are beyond the scope of this section.) The script waits for a shell prompt (either `%` or `#`, followed by a space), then `send` outputs a command to change the working directory, just as if a user had typed the command interactively. The following `expect` command waits for a prompt signifying that the `cd` command has been processed. Finally, `interact` causes Expect to step out of the way so that the user who invoked the Expect script can now type directly to `ssh`.
Expect can be used for many purposes, such as acting as a scriptable front end to debuggers, mailers, and other command-line-driven programs that don’t have scripting languages of their own. The programs require no changes to be driven by Expect. Expect is also useful for regression testing of interactive programs. Expect can be combined with Tk or other Tcl extensions. For example, by using Tk and Expect together, it is possible to write a graphical front end for an existing interactive application without changing the application.

Expect was created by Don Libes of the National Institute of Standards and Technology. For more information, see the Expect home page (http://expect.nist.gov) and the Expect page on the Tcler's Wiki (http://wiki.tcl.tk/expect).

B.13 Extended Tcl

Extended Tcl (TclX) is a library package that augments the built-in Tcl commands with many additional commands and procedures oriented toward system programming tasks. Here are a few of the most popular features of TclX:

- Access to many additional POSIX system calls and functions such as `fork` and `kill`
- A file-scanning facility with functionality much like that of the `awk` program
- Keyed lists, which provide functionality similar to C structures
- Facilities for debugging, profiling, and program development

Many of the best features of TclX are no longer part of it: they turned out to be so widely useful that they were incorporated into the Tcl core. Among the Tcl features pioneered by TclX are file input and output, TCP/IP socket access, array variables, real arithmetic and transcendental functions, autoloading, and the `incr` and `upvar` commands.

Extended Tcl was created by Karl Lehenbauer and Mark Diekhans. For more information, see the TclX SourceForge page (http://tclx.sourceforge.net) and the TclX page on the Tcler's Wiki (http://wiki.tcl.tk/tclx).

C. Tcl Resources

The following are suggested resources for more information on Tcl/Tk.

C.1 Online Resources

Several online resources are available for Tcl/Tk, including

- Tcl Developer's Xchange, `http://www.tcl.tk`

A good contact point for keeping up to date with the Tcl world. The Tcl Developer's Xchange announces new Tcl releases, upcoming conferences, and other significant Tcl events. It also hosts online versions of the Tcl/Tk reference documentation for current and historical releases, Tcl download links, and Tcl advocacy information.

- Tcler's Wiki, `http://wiki.tcl.tk`

A collaboratively edited collection of Tcl wisdom, tips, and code samples. This site is a treasure trove of Tcl information and should be the first place to check when researching Tcl topics.

- ActiveState Programmer Network (ASPN) Tcl Resources, `http://aspn.activestate.com/ASPN/Tcl`

ActiveState's ASPN provides information on technologies supported by ActiveState, including Tcl. The site hosts several Tcl-related mailing lists, a Tcl "cookbook," and other resources. From this site you can also freely download ActiveTcl, ActiveState's binary build of Tcl, and several popular Tcl extensions.

- TkDocs, `http://www.tkdocs.com`

Mark Roseman's site providing Tk tutorials and documentation. Mark has some excellent tips for cross-platform Tk development, and his tutorials cover Tk not only for Tcl but for other dynamic languages like Perl and Python as well.

- Tcl SourceForge Project, `http://tcl.sourceforge.net`

The source code repository for Tcl/Tk and core extensions.

- The `comp.lang.tcl` Usenet newsgroup

The Usenet newsgroup for the exchange of information about Tcl/Tk and related extensions and applications. The newsgroup is used to answer questions from Tcl/Tk users, to exchange information about bugs and their fixes, and to discuss possible new features in Tcl and Tk. New releases of

Tcl/Tk also are announced on this newsgroup, as are releases of other related extensions and applications. You can access the Usenet newsgroups through dedicated news client applications or through web gateways such as Google Groups (http://groups.google.com).

- The Tcl chatroom

A real-time chat service for discussing Tcl/Tk, currently based on the XMPP protocol. Many experienced Tcl programmers are regular contributors, readily answering questions, solving problems, and debating Tcl/Tk development. The chatroom can be accessed via standard XMPP clients, a dedicated TkChat client (http://tkchat.tcl.tk), and web and IRC gateways. See http://wiki.tcl.tk/1178 on the Tcler's Wiki for the latest information on how to access the Tcl chatroom.

C.2 Books

Several books are available for more information on Tcl/Tk and related topics:

- Butenhof, David R. *Programming with POSIX Threads*. Reading, MA: Addison-Wesley, 1997.
- Flynt, Clif. *Tcl/Tk: A Developer's Guide, Second Edition*. San Francisco: Morgan Kaufmann, 2003.
- Friedl, Jeffrey. *Mastering Regular Expressions, Third Edition*. Sebastopol, CA: O'Reilly Media, Inc., 2006.
- Libes, Don. *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*. Sebastopol, CA: O'Reilly Media, Inc., 1994.
- Welch, Brent, and Ken Jones. *Practical Programming in Tcl and Tk, Fourth Edition*. Upper Saddle River, NJ: Prentice Hall, 2003.